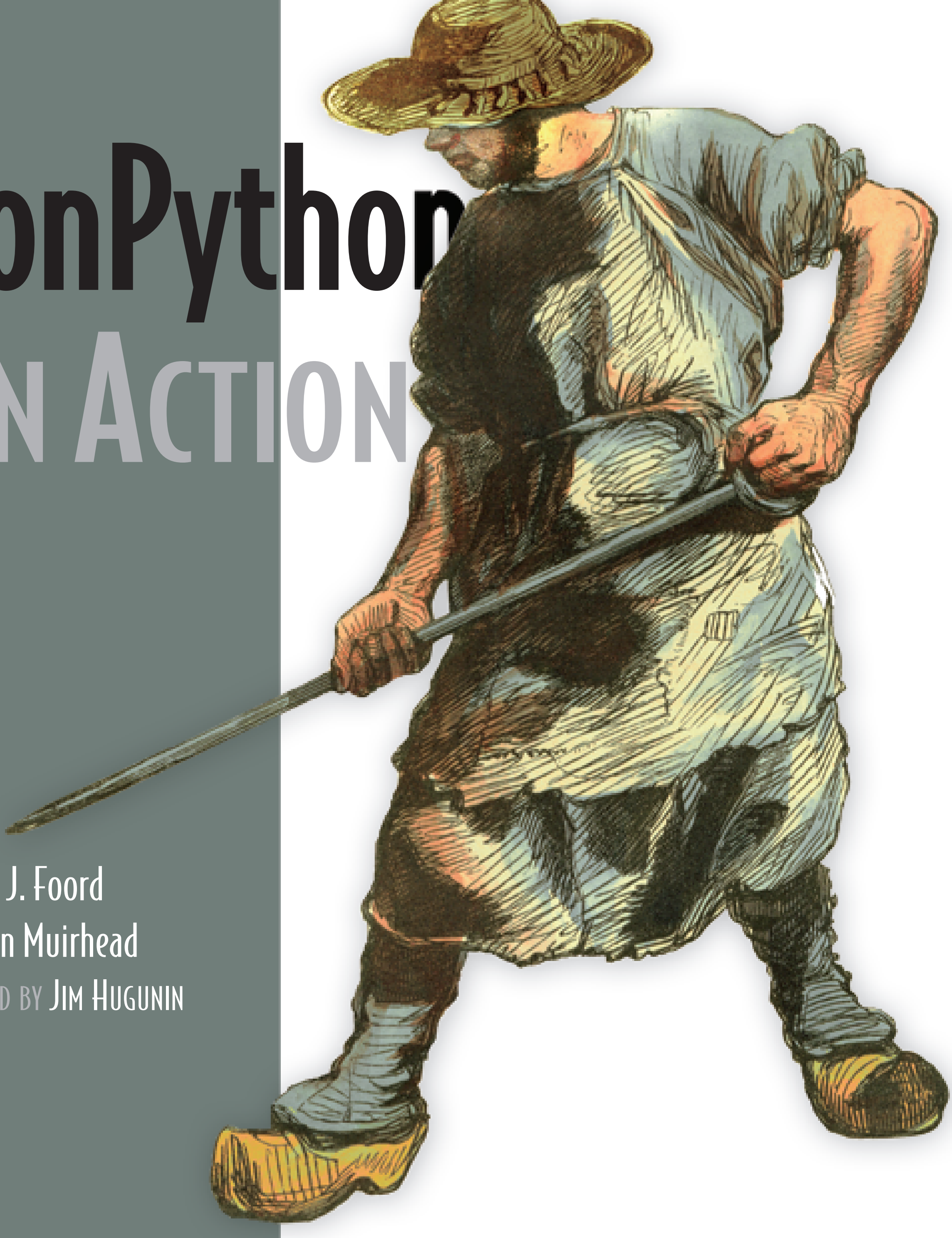
IronPython in Action



Michael J. Foord
Christian Muirhead

Foreword by Jim Hugunin

MANNING

IronPython in Action

IronPython in Action

MICHAEL J. FOORD
CHRISTIAN MUIRHEAD

MANNING
Greenwich
(74° w. long.)

For online information and ordering of this and other Manning books, please visit www.manning.com. The publisher offers discounts on this book when ordered in quantity. For more information, please contact

Special Sales Department
Manning Publications Co.
Sound View Court 3B fax: (609) 877-8256
Greenwich, CT 06830 email: orders@manning.com

Manning Publications Co.
Sound View Court 3B
Greenwich, CT 06830

Development Editor: Jeff Bleil
Copyeditors: Andrea Kaucher, Linda Recktenwald
Typesetter: Gordan Salinovic
Cover designer: Leslie Haimes

ISBN 978-1-933988-33-7
Printed in the United States of America
1 2 3 4 5 6 7 8 9 10 – MAL – 14 13 12 11 10 09

This book is dedicated to the littlest gangster and the mushroom, who endured much throughout its creation.

brief contents

contents

foreword

IronPython brings together two of my favorite things: the elegant Python programming language and the powerful .NET platform.

I've been a fan of the Python language for almost 15 years, ever since it was recommended to me by a fellow juggler while we passed clubs in a park. From the start I found Python to be a simple and elegant language that made it easy to express my ideas in code. I'm amazed by Python's ability to appeal to a broad range of developers, from hard-core hackers to amateur programmers, including scientists, doctors, and animators. I've been teaching my ten-year-old son to program, and even he tells me that "Python is a great language to learn with." Beyond teaching my son, I've tried to contribute to the Python community that gave me this great language and continues to drive it forward. Prior to IronPython, I started the Numeric Python and Jython open source projects.

It took a bit longer for me to become a fan of Microsoft's .NET platform and the Common Language Runtime (CLR) that forms its core execution engine. I first learned about the CLR by reading countless reports on the web that said it was a terrible platform for dynamic languages in general and for Python in particular. IronPython started life as a series of quick prototypes to help me understand how this platform could be so bad. My plan was to prototype for a couple of weeks and then write a pithy paper titled, "Why the CLR is a terrible platform for dynamic languages." This plan was turned upside down when the prototypes turned out to run very well—generally quite a bit faster than the standard C-based Python implementation.

After getting over my initial skepticism, I've grown to love the CLR and .NET as much as Python. While no platform is perfect, this is the closest we've ever come to a universal runtime that can cleanly support a variety of different programming languages. Even more exciting to me is that the team is committed to the multi-language story and we've got great projects like the DLR, IronRuby, and F# to keep extending the range of languages that can coexist on this platform. I've even grown to like C# as the most enjoyable and versatile statically typed programming language I've used.

As the architect for IronPython, I like to believe that it's such a simple and elegant combination of the Python language and the .NET platform that it needs no documentation. After all, who could possibly not know that they should use `clr.Reference` to pass an out parameter to a .NET method? I guess that it's assumptions like that one that would make me a poor choice for writing a book teaching people about IronPython. The best choice for writing a book like this would be a long-term user who's deeply engaged with the community and who has been trying to understand and explain the system to others for years. Now, if only we could find such a person…

I first met Michael Foord in July of 2006. I was preparing an IronPython talk for the OSCON conference in Portland, Oregon. This was going to be an exciting talk where I'd announce that the final release of IronPython 1.0 was weeks away. This was a terrible time to be preparing a talk since my mind and time were occupied with all the details of the actual release. To further complicate things, this was the Open Source Convention, and I knew that I needed to show IronPython running on Linux in order to have credibility with this audience. Unfortunately, I didn't have the time to set up a Linux box and get some useful demos running. Oddly enough, my coworkers (at Microsoft) didn't have any spare Linux boxes running in their offices that I could borrow for a few screen shots.

I did a desperate internet search for "IronPython Linux" and one of the places that led me to was a blog called voidspace. There I found a tutorial on how to use Windows Forms with IronPython. The reason this tutorial showed up was that it included screen caps of the samples running under both Windows and Linux. This was just what I was looking for! By stealing these pictures for my talk I could show people IronPython running on Linux and also point them to an excellent online tutorial to help them learn more about using IronPython than I could cover in a 45-minute talk.

I had a few hesitations about including this reference in my talk. I didn't know anything about the author except that his screen name was Fuzzyman and that he had a personal blog that was subtitled, "the strange and deluded ramblings of a rather odd person." However, I really liked the simple tutorial and I was incredibly happy to have some nice Linux samples to show the OSCON crowd. I was most grateful at the time to this person that I'd never met for helping me out of this jam.

Fuzzyman turned out to be Michael Foord and one of the authors of the book you have in your hands now. Since that first online tutorial, Michael has been helping people to use IronPython through more online samples, presentations at conferences, and through active contributions to the IronPython users mailing list. I couldn't think

of a better way for you to learn how to get started and how to get the most out of IronPython than by following along with Michael and Christian in *IronPython in Action*.

I've spent my career building programming languages and libraries targeted at other developers. This means that the software I write is used directly by a small number of people and it's hard for me to explain to non-developers what I do. The only reason this kind of stuff has value is because of the useful or fun programs that other developers build using it. This book should give you everything you need to get started with IronPython. It will make your development more fun—and more productive. Now go out and build something cool!

JIM HUGUNIN
SOFTWARE ARCHITECT
FOR THE .NET FRAMEWORK TEAM
AND CREATOR OF IRONPYTHON

preface

A programming language is a medium of expression.

—Paul Graham

Neither of us intended to develop with IronPython, least of all write a book on it. It sort of happened by accident. In 2005 a startup called Resolver Systems[1] set up shop in London. They were creating a spreadsheet platform to tackle the myriad problems caused in business by the phenomenal success of spreadsheets. The goal was to bring the proven programming principles of modularity, testability, and maintainability to the spreadsheet world—and having an interpreted language embedded in Resolver One was a core part of this. As Resolver One was to be a desktop application used by financial organizations, it needed to be built on established and accepted technologies, which for the desktop meant .NET.

At the time the choice of interpreted language engines for .NET was limited; even IronPython was only at version 0.7. The two developers who comprised Resolver Systems[2] evaluated IronPython and discovered three important facts:

- Although neither of them was familiar with Python, it was an elegant and expressive language that was easy to learn.
- The .NET integration of IronPython was superb. In fact it seemed that everything they needed to develop Resolver One was accessible from IronPython.
- As a dynamic language, Python was orders of magnitude easier to test than languages they had worked with previously. This particularly suited the test-driven approach they were using.

[1] See http://www.resolversystems.com/ .

[2] Giles Thomas, who is CEO and CTO, and William Reade, a hacker with a great mind for complex systems.

They decided to prototype Resolver One with IronPython, expecting to have to rewrite at least portions of the application in C# at some point in the future. Three years later, Resolver One is in use in financial institutions in London, New York, and Paris; and consists of 40,000 lines of IronPython code[3] with a further 150,000 in the test framework. Resolver One has been optimized for performance several times, and this has always meant fine tuning our algorithms in Python. It hasn't (yet) required even parts of Resolver One to be rewritten in C#.

We are experienced Python developers but neither of us had used IronPython before. We joined Resolver Systems in 2006 and 2007, and were both immediately impressed by the combination of the elegance of Python with the power and breadth of the .NET framework.

Programming is a creative art. Above all Python strives to empower the programmer. It emphasizes programmer productivity and readability, instead of optimizing the language for the compiler. We're passionate about programming, and about Python. In 2007 one of us (Michael) set up the IronPython Cookbook[4] to provide concrete examples for the newly converging IronPython community. Shortly afterwards the two of us decided to write a book that would help both Python and .NET programmers take advantage of all that IronPython has to offer.

[3] With perhaps as many as three hundred lines of C# in total.
[4] At http://www.ironpython.info/ and still an excellent resource!

acknowledgments

Writing this book has been a labor of love for the past two years. One thing that has astonished us is the sheer number of people who are involved in such an endeavor, and how many individuals have helped us. Our thanks for support and assistance go out to our colleagues at Resolver Systems, the team at Manning, our reviewers, virtually the whole IronPython team who gave their advice and support at various times, and all those who bought the Early Access edition and gave feedback and pointed out typos.

These reviewers took time out of their busy schedules to read the manuscript at various times in its development and to send us their input. It is a much better book as a result. Thanks to Leon Bambrick, Max Bolingbroke, Dave Brueck, Andrew Cohen, Dr. Tim Couper, Keith Farmer, Noah Gift, Clint Howarth, Denis Kurilenko, Alex Martelli, Massimo Perga, and Robi Sen.

Without the help of these people, and more, this book wouldn't have been possible. At Manning Publications, Michael Stephens gave us the opportunity, Jeff Bleiel was our tireless editor, Andrea Kaucher and Linda Recktenwald transformed the book through their copyediting, and Katie Tennant did the final proofread. Dino Viehland was our technical editor, and did great work. We also had help from Jimmy Schementi reviewing the Silverlight chapter and from Srivatsn Narayanan on chapter 14.

Special thanks to Jonathan Hartley, a fellow Resolver One hacker, who did a wonderful job producing the figures for *IronPython in Action* and to Jim Hugunin, the creator of IronPython, for writing the foreword.

Michael Foord would also like to express his gratitude to Andrew Lantsbery, for his friendship and technical expertise that proved invaluable.

about this book

IronPython is a radical project for Microsoft. It is the first project to be released under their Ms-PL (Microsoft Public License) open source license. It is also a radically different language from the ones that Microsoft has traditionally promoted for the .NET framework. IronPython is an implementation of the popular programming language Python for .NET. Python is an open source, object-oriented, dynamically typed language in use by organizations like Google, NASA and Pixar. Python is a multi-paradigm language, and brings new possibilities to .NET programmers: not just the added flexibility of dynamic typing, but programming styles such as functional programming and metaprogramming. For Python programmers the powerful runtime, with its JIT compiler and huge range of .NET libraries, also presents new opportunities.

The goal of *IronPython in Action* is not just to teach the mechanics of using IronPython, but to demonstrate the power and effectiveness of object-oriented programming in the Python language. To this end we cover best practices in API design, testing, and the use of design patterns in structured application development. In part this is to dispel the myth that dynamic languages are merely scripting languages; but mostly it is to help you make the best of the language and the platform on which it runs.

The addition of Python to the range of languages available as first-class citizens in .NET reflects the changes happening in the wider world of programming. No one says it better than Anders Hejlsberg, the architect of C#, when asked by Computer World[5] what advice he had for up-and-coming programmers:

[5] See http://www.computerworld.com.au/index.php/id;1149786074;pp;8.

> *Go look at dynamic languages and meta-programming: those are really interesting concepts. Once you get an understanding of these different kinds of programming and the philosophies that underlie them, you can get a much more coherent picture of what's going on and the different styles of programming that might be more appropriate for you with what you're doing right now.*
>
> *Anyone programming today should check out functional programming and meta-programming as they are very important trends going forward.*

Who should read this book?

IronPython in Action is particularly aimed at two types of programmers: Python programmers looking to take advantage of the power of the .NET framework or Mono for their applications, and .NET programmers interested in the flexibility of dynamic languages. It assumes no experience of either Python or .NET, but does assume some previous programming experience. If you have some programming experience, but have never used either of these systems, you should find *IronPython in Action* an accessible introduction to both Python and .NET.

Just as Python is suited to an enormous range of problem domains, so is IronPython. The book covers a range of different uses of IronPython: from web development to application development, one-off scripting to system administration, and embedding into .NET applications for extensible architectures or providing user scripting.

Roadmap

This book contains 15 chapters organized into four parts.

Part 1 Getting started with IronPython—The first part of the book introduces the fundamental concepts behind developing with IronPython and the .NET framework. Chapter 1 introduces IronPython along with key points of interest for both Python and .NET programmers. It finishes by diving into IronPython through the interactive interpreter; a powerful tool for both Python and IronPython. Chapter 2 is a Python tutorial, including areas where IronPython is different from the standard distribution of Python known as CPython. Where chapter 2 is particularly valuable to programmers who haven't used Python before, chapter 3 is an introduction to the .NET framework. As well as covering the basic .NET types (classes, enumerations, delegates, and the like), this chapter shows how to use them from IronPython, ending with a more fully featured "Hello World" program than created in chapter 1.

Part 2 Core development techniques—The next part extends your knowledge of the Python language and the classes available in the .NET framework. It does this by demonstrating a structured approach to Python programming by developing the MultiDoc application using several common design patterns. Figure 1 shows MultiDoc as it looks by the end of chapter 6. Along the way we'll work with Windows Forms, lambdas, properties, decorators, XML, first-class functions, and using C# class libraries created in Visual Studio.

This part finishes by covering testing techniques, to which dynamic languages are especially suited, and some more advanced Python programming techniques such as metaprogramming. The end of chapter 8 contains valuable information about how IronPython interacts with aspects of the Common Language Runtime, information that neither experience with Python nor another .NET framework language alone will furnish you with.

Part 3 IronPython and advanced .NET—The third part takes IronPython into practical and interesting corners of .NET. Each chapter in this part takes an area of .NET programming and shows how best to use it from IronPython.

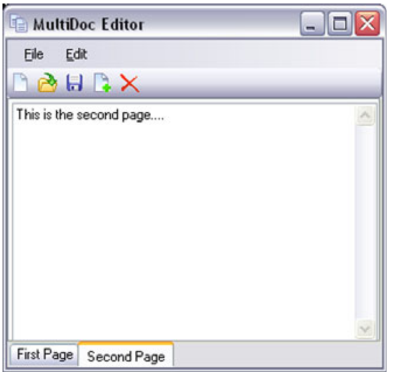


Figure 1 The MultiDoc application as it appears in part 2

- Chapter 9—Writing desktop applications using the Windows Presentation Foundation user interface library
- Chapter 10—System administration, including shell scripting, WMI, and PowerShell
- Chapter 11—Web development with ASP.NET
- Chapter 12—Databases and web services
- Chapter 13—Silverlight

Part 4 Reaching out with IronPython—The final part of this book takes IronPython out into the wilds of a polyglot programming environment. Chapter 14 shows how to create classes in C# and VB.NET for use from IronPython. Of special importance here is creating APIs that feel natural when used from Python, or even giving your objects dynamic behavior. Chapter 15 reverses the situation and embeds IronPython into .NET applications. It tackles the interesting and challenging problem of using dynamic objects from statically typed languages like C# and VB.NET. For many .NET programmers, being able to embed IronPython into applications, to provide a ready-made scripting solution, is the main use case for IronPython.

There are also three appendixes. Appendix A covers the basics of C# and explains the core concepts of the language. Appendix B shows how to create your own objects in Python by implementing its protocol methods. Appendix C has a list of online resources with more information about IronPython and dynamic languages on the .NET framework.

Code conventions and downloads

This book includes copious numbers of examples in Python, C#, and VB.NET. Source code in listings, or in text, is in a `fixed-width font` to separate it from ordinary text. Additionally, method names in text are also presented using `fixed-width font`.

C# and VB.NET can be quite verbose, but even Python is not immune to the occasional long line. In many cases, the original source code (available online) has been reformatted, adding line breaks to accommodate the available page space in the book. In rare cases, even this was not enough, and listings will include line continuation markers. Additionally, comments in the source code have been removed from the listings.

Code annotations accompany many of the source code listings, highlighting important concepts. In some cases, numbered bullets link to explanations that follow the listing.

IronPython is an open source project, released under the very liberal Ms-PL software license. IronPython is available for download, in source or binary form, from the IronPython home page: www.codeplex.com/IronPython.

The source code for all examples in this book is available from Manning's web site: www.manning.com/foord. It is also available for download from the book's website: www.ironpythoninaction.com/.

Author Online

The purchase of *IronPython in Action* includes free access to a private web forum run by Manning Publications, where you can make comments about the book, ask technical questions, and receive help from the authors and from other users. To access the forum and subscribe to it, point your web browser to www.manning.com/ironpythoninaction. This page provides information on how to get on the forum once you are registered, what kind of help is available, and the rules of conduct on the forum.

Manning's commitment to our readers is to provide a venue where a meaningful dialogue between individual readers and between readers and the author can take place. It is not a commitment to any specific amount of participation on the part of the author, whose contribution to the forum remains voluntary (and unpaid). We suggest you try asking the authors some challenging questions lest their interest stray! The Author Online forum and the archives of previous discussions will be accessible from the publisher's web site as long as the book is in print.

About the authors

Michael Foord and Christian Muirhead both work full time with IronPython for Resolver Systems, creating a highly programmable spreadsheet called Resolver One. They have been using IronPython since before version 1.0 was released.

Michael Foord has been developing with Python since 2002. He blogs and writes about Python and IronPython far more than is healthy for one individual and in 2008 was made the first Microsoft MVP for dynamic languages. As the Resolver Systems community champion he speaks internationally on Python and IronPython. He maintains the IronPython Cookbook[6] and IronPython-URLs[7] websites, and can also be found online at http://www.voidspace.org.uk/python/weblog/. In the real world he lives in Northampton, UK, with his wife Delia.

6 See http://www.ironpython.info/.
7 See http://ironpython-urls.blogspot.com/.

Christian Muirhead began his career in a high-volume database environment, and for the last eight years has been building database-driven websites. He has five years of experience working with C#, the .NET framework, and ASP.NET. He has been using Python in most of his projects since discovering it in 1999, including building web applications for the BBC using Django. Christian is a New Zealander currently exiled in London with his wife Alice.

About the title

By combining introductions, overviews, and how-to examples, the *In Action* books are designed to help learning and remembering. According to research in cognitive science, the things people remember are things they discover during self-motivated exploration.

Although no one at Manning is a cognitive scientist, we are convinced that for learning to become permanent it must pass through stages of exploration, play, and, interestingly, re-telling of what is being learned. People understand and remember new things, which is to say they master them, only after actively exploring them. Humans learn *in action*. An essential part of an *In Action* guide is that it is example-driven. It encourages the reader to try things out, to play with new code, and explore new ideas.

There is another, more mundane, reason for the title of this book: our readers are busy. They use books to do a job or solve a problem. They need books that allow them to jump in and jump out easily and learn just what they want just when they want it. They need books that aid them *in action*. The books in this series are designed for such readers.

About the cover illustration

The caption of the figure on the cover of *IronPython in Action* reads "An Ironworker." The illustration is taken from a French book of dress customs, *Encyclopedie des Voyages* by J. G. St. Saveur, published in 1796. Travel for pleasure was a relatively new phenomenon at the time and illustrated guides such as this one were popular, introducing both the tourist as well as the armchair traveler to the inhabitants of other regions of the world, as well as to the regional costumes and uniforms of French soldiers, civil servants, tradesmen, merchants, and peasants.

The diversity of the drawings in the *Encyclopedie des Voyages* speaks vividly of the uniqueness and individuality of the world's towns and provinces just 200 years ago. This was a time when the dress codes of two regions separated by a few dozen miles identified people uniquely as belonging to one or the other, and when members of a social class or a trade or a profession could be easily distinguished by what they were wearing.

Dress codes have changed since then and the diversity by region and social status, so rich at the time, has faded away. It is now often hard to tell the inhabitant of one continent from another. Perhaps, trying to view it optimistically, we have traded a cultural and visual diversity for a more varied personal life. Or a more varied and interesting intellectual and technical life.

We at Manning celebrate the inventiveness, the initiative, and the fun of the computer business with book covers based on the rich diversity of regional life two centuries ago brought back to life by the pictures from this travel guide.

Part 1

Getting started with IronPython

Like all good books, and possibly a few bad ones, this one starts with an introduction. In this section, we cover what IronPython is, how it came into being, and why a language like Python is a big deal for .NET. You'll also get to use the IronPython interactive interpreter, which is both a powerful tool and a great way of showing off some of the features of Python. Chapter 2 is a swift tutorial for the Python language. It won't make you a Python master, but it will prepare you for the examples used throughout this book, and serve as a useful reference well beyond. Chapter 3 briefly introduces .NET and then wades into programming with IronPython, taking Windows Forms as the example. While gaining an understanding of concepts essential to any real work with IronPython, you'll be getting your hands dirty with some real code. First, though, let's discuss how IronPython fits in with .NET.

1 A new language for .NET

This chapter covers

- An introduction to IronPython
- Python and dynamic languages on .NET
- The IronPython interactive interpreter
- Live object introspection with help and dir

The .NET framework was launched in 2000 and has since become a popular platform for object-oriented programming. Its heart and soul is the Common Language Runtime (CLR), which is a powerful system including a just-in-time compiler, built-in memory management, and security features. Fortunately, you can write .NET programs that take advantage of many of these features without having to understand them, or even be aware of them. Along with the runtime comes a vast array of libraries and classes, collectively known as the framework classes. Libraries available in the .NET framework include the Windows Forms and Windows Presentation Foundation (WPF)[1] graphical user interfaces, as well as

[1] Microsoft's next generation user interface framework.

libraries for communicating across networks, working with databases, creating web applications, and a great deal more.

The traditional languages for writing .NET programs are Visual Basic.NET, C#, and C++.[2] IronPython is a .NET compiler for a programming language called Python, making IronPython a first-class .NET programming language. If you're a .NET developer, you can use Python for tasks from web development to creating simple administration scripts, and just about everything in between. If you're a Python programmer, you can use your favorite language to take advantage of the .NET framework.

IronPython isn't cast in the same mold as traditional .NET languages, although there are similarities. It's a dynamically typed language, which means a lot of things are done differently and you can do things that are either impossible or more difficult with alternative languages. Python is also a multi-paradigm language. It supports such diverse styles of programming as procedural and functional programming, object-oriented programming, metaprogramming, and more.

Microsoft has gone to a great deal of trouble to integrate IronPython with the various tools and frameworks that are part of the .NET family. They've built specific support for IronPython into the following projects:

- *Visual Studio*—The integrated development environment
- *ASP.NET*—The web application framework
- *Silverlight*—A browser plugin for client-side web application programming
- *XNA*[3]—The game programming system
- *Microsoft Robotics Kit*—An environment for robot control and simulation
- *Volta*—An experimental recompiler from Intermediate Language bytecode (IL) to JavaScript[4]
- *C# 4.0*—The next version of C# and the CLR that will include dynamic features using the Dynamic Language Runtime (DLR)

IronPython is already being used in commercial systems, both to provide a scripting environment for programs written in other .NET languages and to create full applications. One great example called Resolver One,[5] a spreadsheet development environment, is how I (Michael) got involved with IronPython. You can see a screenshot of Resolver One in figure 1.1. At last count, there were over 40,000 lines of IronPython code in Resolver One, plus around 150,000 more in the test framework developed alongside it.

By the end of *IronPython in Action*, we hope you'll have learned everything you need to tackle creating full applications with IronPython, integrating IronPython as part of another application, or just using it as another tool in your toolkit. You'll also have

---

2 In the C++/CLI flavor, which is sometimes still referred to by the name of its predecessor, Managed C++. Use of C# and VB.NET is more widespread for .NET programming.

3 XNA is a recursive acronym standing for XNA's Not Acronymed.

4 Allowing you to write client-side code for web applications in Python and have it recompiled to JavaScript for you.

5 See http://www.resolversystems.com.

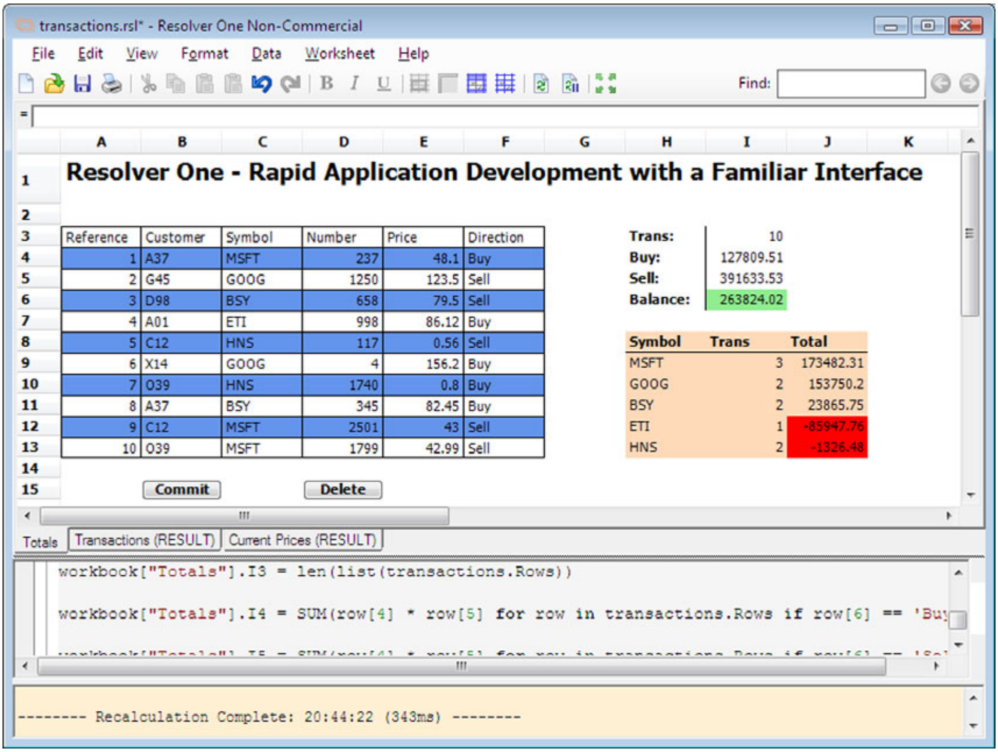


Figure 1.1 Resolver One: A full application written in IronPython

explored some of these alternative programming techniques and used a variety of different aspects of the .NET framework. This exploration will enable you to make the best use of the Python language and the wealth of classes made available by .NET.

Before we can achieve any of this, an introduction is in order. This chapter introduces IronPython and the Python programming language. It explains why Python is a good fit for the .NET framework and will give you a tantalizing taste of what is possible with IronPython, via the interactive interpreter.

1.1 An introduction to IronPython

Python is a dynamic language that has been around since 1990 and has a thriving user community. Dynamic languages don't require you to declare the type of your objects, and they allow you greater freedom to create new objects and modify existing ones at runtime. On top of this, the Python philosophy places great importance on readability, clarity, and expressiveness. Figure 1.2 is a slide from a

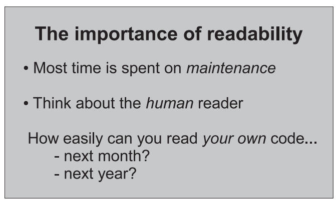


Figure 1.2 A slide from a presentation, emphasizing a guiding philosophy of Python

presentation[6] by Guido van Rossum, the creator of Python; it explains why readability is so important in Python.

IronPython is an open source implementation of Python for .NET. It has been developed by Microsoft as part of making the CLR a better platform for dynamic languages. In the process, they've created a fantastic language and programming environment. But what exactly is IronPython?

1.1.1 What is IronPython?

IronPython primarily consists of the IronPython engine, along with a few other tools to make it convenient to use. The IronPython engine compiles Python code into IL, which runs on the CLR. Optionally IronPython can compile to assemblies, which can be saved to disk and used to make binary-only distributions of applications.

> **Assemblies**
>
> Assemblies are .NET libraries or executables. .NET consists of a great deal of these assemblies, in which the framework classes live, in the form of dlls.
>
> Because of the memory management and security features that .NET provides, code in .NET assemblies is called *managed code*.[7]
>
> Assemblies contain code compiled from .NET languages into Intermediate Language (IL) bytecode. IL is run with the just-in-time (JIT) compiler for fast execution.

You can see how Python code is compiled and run by the IronPython engine in figure 1.3.

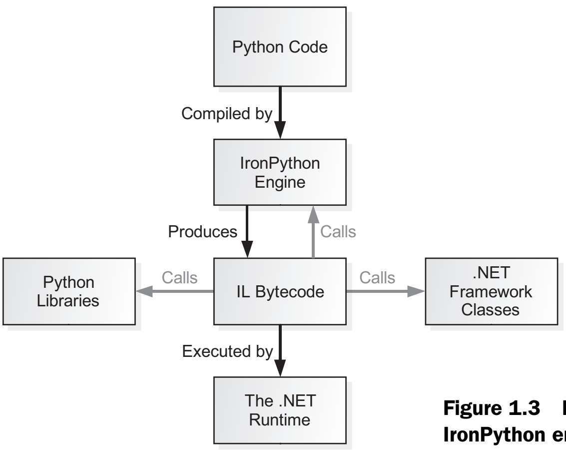


Figure 1.3 How Python code and the IronPython engine fit into the .NET world

[6] See http://www.python.org/doc/essays/ppt/hp-training/index.htm.
[7] .NET does provide ways to access unmanaged code contained in traditional compiled dlls.

Figure 1.3 shows the state of IronPython version 1.[8] In April 2007, the IronPython team released an early version of IronPython 2, which introduces a radical new development, the Dynamic Language Runtime (DLR). The DLR is a hosting platform and dynamic type system taken out of IronPython 1 and turned into a system capable of running many different dynamic languages. You'll be hearing more about the DLR in a short while.

Because Python is a highly dynamic language, the generated assemblies remain dependent on the IronPython dlls. Despite this, they're still only compiled .NET code, so you can use classes from the .NET framework directly within your code without needing to do any type conversions yourself.

Accessing the .NET framework from IronPython code is extremely easy. As well as being a programming language in its own right, IronPython can be used for all the typical tasks you might approach with .NET, such as web development with ASP.NET (Active Server Pages, the .NET web application framework) or creating smart client applications with Windows Forms or WPF. As an added bonus, IronPython also runs on the version of the CLR shipped with Silverlight 2. You can use IronPython for writing client-side applications that run in a web browser, something that Python programmers have wanted for years!

IronPython itself is written in C# and is a full implementation of Python. IronPython 1 is Python version 2.4, whereas IronPython 2 is Python 2.5. If you've used Python before, IronPython is Python with none of the core language features missing or changed. Let's make this clear: IronPython *is* Python.

Development cycles are typically fast with Python. With dynamically typed languages, tasks can be achieved with less code, making IronPython ideal for prototyping applications or scripting system administration tasks that you can't afford to spend a lot of time on. Because of the readability and testability of well-written Python code, it scales well to writing large applications. You are likely to find that your prototypes or scripts can be refactored into full programs much more easily than writing from scratch in an alternative language.

If you're already developing with .NET, you needn't do without your favorite tools. Microsoft has incorporated IronPython support into Visual Studio 2005 through the Software Development Kit (SDK).[9] You can use Visual Studio to create IronPython projects with full access to the designer and debugger. Figure 1.4 shows Visual Studio being used to create a Windows application with IronPython.

Visual Studio 2008 integration exists in the form of IronPython Studio,[10] which is implemented through the Visual Studio Shell extensibility framework. IronPython

[8] And as a simplified view, it's true of IronPython 2 as well, except the IronPython engine is comprised of the Dynamic Language Runtime *and* IronPython-specific assemblies.

[9] The Visual Studio SDK is a Microsoft extension that includes IronPython support.

[10] http://www.codeplex.com/IronPythonStudio

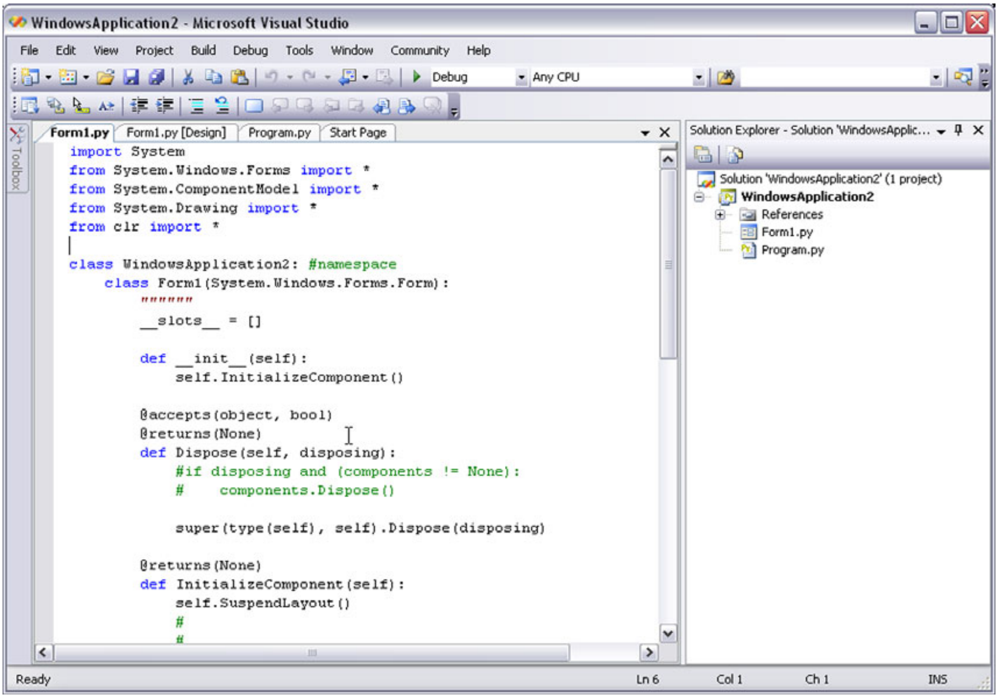


Figure 1.4 Generated IronPython code in Visual Studio

Studio can either be run standalone (without requiring Visual Studio to be installed) or integrated into Visual Studio. It includes Windows Forms and WPF designers and is capable of producing binary executables from Python projects. Figure 1.5 shows IronPython Studio running in integrated mode as part of Visual Studio 2008.

An alternative version of .NET called Mono provides a C# compiler, runtime, and a large proportion of the framework for platforms other than Windows. IronPython runs fine on Mono, opening up the possibility of creating fully featured cross-platform programs using IronPython. Windows Forms is available on Mono, so GUI applications written with IronPython can run on any of the many platforms that Mono works on.

IronPython is a particularly interesting project for Microsoft to have undertaken. Not only have they taken a strong existing language and ported it to .NET, but they have chosen to release it with a sensible open source license. You have full access to IronPython's source code, which is a good example of compiler design, and you can create derivative works and release them under a commercial license. This open approach is at least partly due to the man who initiated IronPython, Jim Hugunin. Let's explore his role in creating IronPython, along with a brief history lesson.

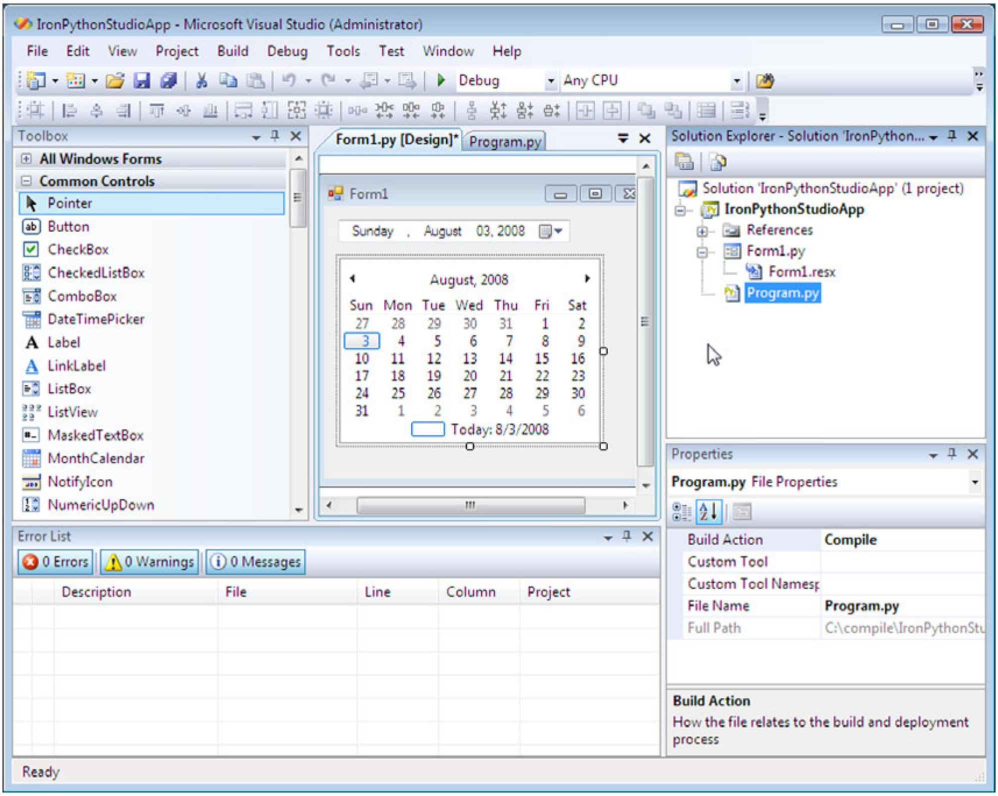


Figure 1.5 Using the Windows Forms designer with IronPython Studio running in Visual Studio 2008

1.1.2 *A brief history of IronPython*

The standard version of Python is often referred to as CPython, usually in the context of distinguishing it from other implementations; the C is because it's written in C. CPython is overwhelmingly the most-used version of Python, and most Python code is written to run on it. CPython isn't Python, though. Python is a programming language, and CPython is only one implementation (albeit an important one).[11]

IronPython isn't the first version of Python to target an alternative platform to the usual Python runtime. The most famous alternative is Jython, Python for the Java Virtual Machine (JVM). The original version of Jython (or JPython, as it was known then) was created by a gentleman called Jim Hugunin.

Over the last few years, dynamic languages have been rising in popularity. Their emphasis on concise code and empowering the programmer have attracted a great deal of developer interest. But back in 2003, the CLR was widely regarded as being a

---

[11] Python has no formal specification. It's defined by the language reference documentation and from CPython, which is called a reference implementation.

Python implementations

The most common Python implementation is called CPython. Other implementations include the following:

IronPython—For .NET.

Jython—For the Java VM.

PyPy—An experimental interpreter compiler toolchain with a multitude of backends (target platforms). It includes an implementation of Python in Python.

Stackless—An alternative to CPython that makes minimal use of the C stack and has support for green threads.

tinypy—A minimal implementation of Python in 64KB of code. Useful for embedded systems.

bad platform for hosting dynamic languages.[12] Jim decided to write an article examining why .NET was so bad for these languages.

His experience with the JVM proved that it was certainly *possible* to create language runtimes capable of hosting both static and dynamic languages, and he wondered what Microsoft had gotten so wrong. Naturally he explored this by attempting a toy implementation of Python. To his horror, he discovered that, contrary to popular opinion, Python worked well on .NET. In fact, his initial attempt ran the basic Python benchmark pystone 1.7 times faster than CPython.

This outcome was unfortunate because a full language implementation is a major undertaking, and Jim now felt honor bound to take his experiment further.

After making his results public, Jim was invited to present them to Microsoft. Microsoft was particularly interested in the challenges and difficulties that Jim had encountered because they were keen to make the CLR a better platform for dynamic languages.

The upshot is that Jim joined the CLR team at Microsoft. A group of programmers were brought together to work on IronPython and, in the process, help improve the CLR. Importantly, Microsoft agreed to keep IronPython open source, with a straightforward license similar to the BSD[13] license.

Since the early releases, the Python community has been closely involved in the development of IronPython. Releases were made often, and Python programmers have been quick[14] to point out bugs in IronPython, or differences between IronPython and

[12] For example, see the InfoWorld article from 2004, "Does .Net have a dynamic-language deficiency?" at http://www.infoworld.com/article/04/02/27/09FEmsnetdynamic_1.html. Ironically, this was written by Jon Udell, who now works for Microsoft.

[13] The BSD license is a popular (and permissive) open source license that originated with the Berkeley Software Distribution, a Unix-like operating system.

[14] At least perhaps partly because of suspicions about Microsoft's intentions for Python…

the way CPython behaves. The IronPython team was (and, in fact, still is) both fast and scrupulous in fixing bugs and incompatibilities between CPython and IronPython.

After many beta releases, IronPython 1.0 Production was released in September 2006. Meanwhile, other Microsoft teams were working to ensure that IronPython fit into the different members of the .NET family, including a Community Technology Preview (CTP) called IronPython for ASP.NET. IronPython for ASP.NET enables IronPython to be used for .NET web development, and introduces a change to the normal ASP model called *no-compile pages*.

Then in spring 2007, Microsoft surprised just about everyone with two important releases. The first was an alpha version of IronPython 2. IronPython 2 is built on top of an important component called the Dynamic Language Runtime (DLR).

The second surprise announcement, following hot on the heels of the DLR, was the release of Silverlight. Silverlight is a plugin that runs inside the browser for animation, video streaming, or creating rich-client web applications. The biggest surprise was that Silverlight 2 includes a cut-down version of .NET, called the CoreCLR, and the DLR can run on top of it. Any of the languages that use the DLR can be used for client-side web programming. The Python community in particular has long wanted a secure version of Python that they could use for client-side web programming. The last place they expected it to come from was Microsoft!

We're hoping that you're already convinced that IronPython is a great programming language; but, as a developer, why should you want to use IronPython? There are two types of programmers for whom IronPython is particularly relevant. The first is the large number of Python programmers who now have a new implementation of Python running on a platform different than the one they're used to. The second type is .NET programmers, who might be interested in the possibilities of a dynamic language or perhaps need to embed a scripting engine into an existing application. We take a brief look at IronPython from both these perspectives, starting with Python programmers.

1.1.3 IronPython for Python programmers

As we've mentioned before, IronPython is a full implementation of Python. If you've already programmed with Python, there's nothing to stop you from experimenting with IronPython immediately.

The important question is, why would a Python programmer be interested in using IronPython? The answer is twofold: the platform and the platform. We know that, initially, that might not make much sense, but bear with us. We're referring first to the underlying platform that IronPython runs on—the CLR. Second, along with the runtime comes the whole .NET framework, a huge library of classes a bit like the Python standard library.

The CLR is an interesting platform for several reasons. The CLR has had an enormous amount of work to make it fast and efficient. Multithreaded programs can take full advantage of multiple processors, something that CPython programs can't do

because of a tricky creature called the GIL.[15] Because of the close integration of IronPython with the CLR, extending IronPython through C# code is significantly easier than extending CPython with C. There's no C API to contend with; you can pass objects back and forth across the boundary without hassles and without reference counting[16] to worry about. On top of all this, .NET has a concept called AppDomains. AppDomains allow you to run code with reduced privileges, such as preventing it from accessing the filesystem, a feature that has long been missing from CPython.

NOTE The ability to take advantage of multicore CPUs within a single process and the no-hassles bridge to C# are two of the major reasons that a Python programmer should be interested in IronPython. Chapter 14 shows how easy it is to extend IronPython from C#.

IronPython programs use .NET classes natively and seamlessly, and there are lots of classes. Two of the gems in the collection are Windows Forms and WPF, which are excellent libraries for building attractive and native-looking user interfaces. As a Python programmer, you may be surprised by how straightforward the programmer's interface to these libraries feels. Whatever programming task you're approaching, it's likely that there's some .NET assembly available to tackle it. As well as the standard libraries that come with the framework, there are also third-party libraries for sophisticated GUI components, such as data grids, that don't have counterparts in CPython.

Table 1.1 shows a small selection of the libraries available to you in the .NET framework.

Table 1.1 Common .NET assemblies and namespaces

Assembly/namespace name	Purpose
`System`	Contains the base .NET types, exceptions, garbage collection classes, and much more.
`System.Data`	Classes for working with databases, both high and low level.
`System.Drawing`	Provides access to the GDI+ graphics system.
`System.Management`	Provides access to Windows management information and events (WMI), useful for system administration tasks.
`System.Environment`	Allows you to access and manipulate the current environment, like command-line arguments and environment variables.
`System.Diagnostics`	Tracing, debugging, event logging, and interacting with processes.
`System.XML`	For processing XML, including SOAP, XSL/T and more.
`System.Web`	The ASP.NET web development framework.

15 The Global Interpreter Lock, which makes some aspects of programming with Python easier but has this significant drawback.

16 CPython uses reference counting for garbage collection, which extension programmers have to take into account.

Table 1.1 Common .NET assemblies and namespaces *(continued)*

Assembly/namespace name	Purpose
`System.IO`	Contains classes for working with paths, files, and directories. Includes classes to read and write to filesystems or data streams, synchronously or asynchronously.
`Microsoft.Win32`	Classes that wrap Win32 common dialogs and components including the registry.
`System.Threading`	Classes needed for multithreaded application development.
`System.Text`	Classes for working with strings (such as `StringBuilder`) and the `Encoding` classes that can convert text to and from bytes.
`System.Windows.Forms`	Provides a rich user interface for applications.
`System.Windows`	The base namespace for WPF, the new GUI framework that's part of .NET 3.0.
`System.ServiceModel`	Contains classes, enumerations, and interfaces to build Windows Communication Foundation (WCF) service and client applications.

As we go through the book, we use several of the common .NET assemblies, including some of those new in .NET 3.0. More importantly, you'll learn how to understand the MSDN documentation so that you're equipped to use *any* assembly from IronPython. We also do some client-side web programming with Silverlight, scripting the browser with Python, something impossible before IronPython and Silverlight.

Most of the Python standard library works with IronPython; ensuring maximum compatibility is something the Microsoft team has put a lot of work into. But beware—not all of the standard library works; C extensions don't work because IronPython isn't written in C. In some cases, alternative wrappers may be available,[17] but parts of the standard library and some common third-party extensions don't work yet. If you're willing to swap out components with .NET equivalents or do some detective work to uncover the problems, it's usually possible to port existing projects.

IronPython shines with new projects, particularly those that can leverage the power of the .NET platform. To take full advantage of IronPython, you'll need to know about a few particular features. These include things that past experience with Python alone hasn't prepared you for. Before we turn to using IronPython, let's first look at how it fits in the world of the .NET framework.

1.1.4 IronPython for .NET programmers

IronPython is a completely new language available to .NET programmers. It opens up new styles of programming and brings the power and flexibility of dynamic programming languages to the .NET platform.

Programming techniques such as functional programming and creating classes and functions at runtime are possible with traditional .NET languages like VB.NET and

[17] Several of these are provided by FePy, a community distribution of IronPython. See http://fepy.sourceforge.net/

C#, but they're a lot easier with Python. You also get straightforward closures, duck typing, metaprogramming, and much more thrown in for free. We explore some of the features that make dynamic languages powerful later in the book.

IronPython is a full .NET language. Every feature of the .NET platform can be used, with the (current) exception of attributes for which you can use stub classes written in C#. All your existing knowledge of the .NET framework is relevant for use with IronPython.

So why would you want to use IronPython? Well, we can suggest a few reasons.

Without a compile phase, developing with IronPython can be a lot quicker than with traditional .NET languages; it typically requires less code and results in more readable code. A fast edit-run cycle makes IronPython ideal for prototyping and for use as a scripting language. Classes can be rewritten in C# at a later date (if necessary) with minimal changes to the programmer's interface.

IronPython may be a new language, but Python isn't. Python is a mature and stable language. The syntax and basic constructs have been worked out over years of programming experience. Python has a clear philosophy of making things as easy as possible for the programmer rather than for the compiler. Common concepts should be both simple and elegant to express, and the programmer should be left as much freedom as possible.

The best way to illustrate the difference between Python and a static language like C# is to show you. Table 1.2 demonstrates a simple Hello World program written in both C# and Python. A `Hello` class is created, and a `SayHello` method prints *Hello World* to the console. Differences (and similarities) between the two languages are obvious.

Table 1.2 Hello World compared in C# and IronPython

<table>
<tr><th>A small Hello World app in C#</th><th>Equivalent in Python</th></tr>
<tr><td><pre>
using System;

class Hello
{
  private string _msg;

  public Hello()
  {
    _msg = "Hello World";
  }

  public Hello(string msg)
  {
    _msg = msg;
  }

  public void SayHello()
  {
     Console.WriteLine(
        _msg);
  }

  public static void Main()
  {
    Hello app = new Hello();
    app.SayHello();
  }
}
</pre></td><td><pre>
class Hello(object):
  def __init__(self,
      msg='hello world'):
    self.msg = msg

    def SayHello(self):
        print self.msg

app = Hello ()
app.SayHello()
</pre></td></tr>
</table>

You can see from this example how much extra code is required in C# for the sake of the compiler. The curly braces, the semicolons, and the type declarations are all line noise and don't add to the functionality of the program. They do serve to make the code harder to read.

This example mainly illustrates that Python is syntactically more concise than C#. It's also *semantically* more concise, with language constructs that allow you to express complex ideas with a minimum of code. Generator expressions, multiple return values, tuple unpacking, decorators, and metaclasses are a few favorite language features that enhance Python expressivity. We explore Python itself in more depth in next chapter.

If you're new to dynamic languages, the interactive interpreter will also be a pleasant surprise. Far from being a toy, the interactive interpreter is a fantastic tool for experimenting with classes and objects at the console. You can instantiate classes and explore their properties live, using introspection and the built-in `dir` and `help` commands to see what methods and attributes are available to you. As well as experimenting with objects, you can also try out language features to see how they work, and the interpreter makes a great calculator or alternative shell. You get a chance to play with the interactive interpreter at the end of this chapter.

If you're looking to create a scripting API for an application, embedding IronPython is a great and ready-made solution. You can provide your users with a powerful and easy-to-learn scripting language that they may already know and that has an abundance of resources for those who want to learn. The IronPython engine and its API have been designed for embedding from the start, so it requires little work to integrate it into applications.

It's time to take a closer look at Python the language and the different programming techniques it makes possible. We even reveal the unusual source of Python's name.

1.2 Python on the CLR

The core of the .NET framework is the CLR. As well as being at the heart of .NET and Mono, it's also now (in a slightly different form) part of the Silverlight runtime.

The CLR runs programs that have been compiled from source code into bytecode. Any language that can be compiled to IL can run on .NET. The predominant .NET languages, VB.NET and C#, are statically typed languages. Python is from a different class of language—it's dynamically typed.

Let's take a closer look at some of the things that dynamic languages have to offer programmers, including some of the language features that make Python a particularly interesting language to work with. We cover both Python the language and a new platform for dynamic languages: Silverlight. We start with what it means for a language to be dynamic.

1.2.1 Dynamic languages on .NET and the DLR

For a while, the CLR had the reputation of being a bad platform for dynamic languages. As Jim Hugunin proved with IronPython, this isn't true. One of the reasons that Microsoft took on the IronPython project was to push the development of the

CLR to make it an even better platform for hosting multiple languages, particularly dynamic languages. The DLR, which makes several dynamic languages available for .NET, is the concrete result of this work.

So, why all the fuss about dynamic languages?

First, dynamic languages are trendy; all the alpha-geeks are using them! Clearly, this isn't enough of an explanation. Unfortunately, like many programming terms, dynamic is hard to pin down and define precisely. Typically, it applies to languages that are dynamically typed, that don't need variable declarations, and where you can change the type of object a variable refers to.

More importantly, you can examine and modify objects at runtime. Concepts such as introspection and reflection, although not exclusive to dynamic languages, are *very* important and are simple to use. Classes, functions, and libraries (called modules in Python), are first-class objects that can easily be created in one part of your code and used elsewhere.

In statically typed languages, method calls are normally bound to the corresponding method at compile time. With Python, the methods are looked up (dynamically) at runtime, so modifying runtime behavior is much simpler. This is called late binding.

With static typing, you must declare the type of every object. Every path through your code that an object can follow must respect that type. If you deviate from this, the compiler will reject the program. In a situation where you may need to represent any of several different pieces of information, you may need to implement a custom class or provide alternative routes through your code that essentially duplicate the same logic.

In dynamic languages, objects still have a type. Python is a strongly typed language,[18] and you can only perform operations that make sense for that type. For example, you can't add strings to numbers. The difference is that the Python interpreter doesn't check types (or look up methods) until it needs to. Any name can reference an object of any type, and you can change the object that a variable points to. This is dynamic typing. Objects can easily follow the same path; you only differentiate on the type at the point where it's relevant. One consequence of this is that container types (lists, dictionaries and tuples, plus user-defined containers in Python) can automatically be heterogeneous. They can hold objects of any type with no need for the added complexity of generics.

In many cases, the type doesn't even matter, as long as the object supports the operation being performed; this is called duck typing.[19] Duck typing can remove the need for type checking and formal interfaces. For example, to make an object indexable as a dictionary-like container, you only need to implement a single method.[20]

All this can make programmers who are used to static type checking nervous. In statically typed languages, the compiler checks types at compile time, and will refuse to compile programs with type errors. This kind of type checking is impossible with

---

[18] Some dynamic languages are weakly typed and allow you to do some very odd things with objects of different types.

[19] If the object walks like a duck and quacks like a duck, let's treat it like a duck...

[20] That method is called `__getitem__` and is used for both the mapping and sequence protocols in Python.

dynamic languages,[21] so type errors can only be detected at runtime. Automated testing is more important with dynamic languages—which is convenient because they're usually easier to test. Dynamic language programmers are often proponents of *strong testing rather than static checking*.[22]

Our experience is that type errors form the minority of programming errors and are usually the easiest to catch. A good test framework will greatly improve the quality of your code, whether you're coding in Python or another language, and the benefits of using a dynamic language outweigh the costs.

Because types aren't enforced at compile time the container types (in Python the list, tuple, set, and dictionary) are heterogeneous—they can contain objects of different types. This can make defining and using complex data structures trivially easy.

These factors make dynamic languages compelling for many programmers; but, before IronPython, they weren't available for those using .NET. Microsoft has gone much further than implementing a single dynamic language, though. With IronPython, Jim and his team proved that .NET was a good environment for dynamic languages, and they created the entire infrastructure necessary for one specific language. In the DLR, they abstracted out of IronPython a lot of this infrastructure so that other languages could be implemented on top of it.

With the DLR it should be *much* easier to implement new dynamic languages for .NET, and even have those languages interoperate with other dynamic languages because they share a common type system. To prove the worth of the DLR, Microsoft has already created three languages that run on the DLR. The first of these is IronPython 2, followed by IronRuby, and Managed JScript. Table 1.3 lists the (current) languages that run on the DLR.

Table 1.3 Languages that run on the DLR

Language	Notes
IronPython 2	The latest version of IronPython. Built on top of the DLR.
IronRuby	A port of the Ruby language to run on .NET. Implemented by John Lam and team.
Managed JScript	An implementation of ECMA 3, otherwise known as JavaScript. Currently only available for Silverlight.
ToyScript[a]	A simple example language. Illustrates how to build languages with the DLR.
IronScheme[b]	An R6RS-compliant Scheme implementation based on the DLR.

a. See http://www.iunknown.com/2007/06/getting_started.html.
b. See http://www.codeplex.com/IronScheme.

21 Although not all statically typed languages require type declarations. Haskell is a statically typed language that uses type inferencing instead of type declarations. ShedSkin is a Python to C++ compiler that also uses type inferencing and compiles a static subset of the Python language into C++. C# 3.0 gained an extremely limited form of type inferencing with the introduction of the `var` keyword.

22 A phrase borrowed from Bruce Eckel, a strong enthusiast of dynamic languages in general and Python in particular. See http://www.mindview.net/WebLog/log-0025.

Importantly, C# 4.0 will include dynamic features through the DLR.[23] This brings late binding and dynamic dispatch to C#, but will also make it easier to interact with dynamic languages from C#.

As an IronPython programmer, you needn't even be aware of the DLR; it's just an implementation detail of how IronPython 2 works. It does become relevant when you're embedding IronPython into other applications.

Another important consequence of the DLR has to do with Silverlight, Microsoft's new browser plugin.

1.2.2 Silverlight: a new CLR

At Mix 2007, a conference for web designers and developers, Microsoft surprised just about everyone by announcing both the DLR *and* Silverlight.

Silverlight is something of a breakthrough for Microsoft. Their usual pattern is to release an early version of a project with an exciting-sounding name, and then the final version with an anonymous and boring name.[24] This time around they've broken the mold; Silverlight was originally codenamed Windows Presentation Foundation Everywhere, or WPF/E. WPF is the new user interface library that's part of .NET 3.0, and WPF/E takes it to the web.

What Silverlight *really* is, is a cross-platform and cross-browser[25] plugin for animation, video streaming, and rich web applications. The animation and video streaming features are aimed at designers who would otherwise be using Flash. What's more exciting for developers is that Silverlight 2 comes with a version of the CLR called the CoreCLR. The CoreCLR is a cut-down .NET runtime containing (as the name implies) the core parts of the .NET framework. You can create web applications working with .NET classes that you're already familiar with. It all runs in a sandboxed environment that's safe for running in a browser.

Although Silverlight doesn't work on Linux, Microsoft is cooperating with the Mono team to produce Moonlight,[26] which is an implementation of Silverlight based on Mono. It will initially run on Firefox on Linux, but eventually will support multiple browsers everywhere that Mono runs.

The best thing about Silverlight is that the DLR will run on it. So not only can you program Silverlight applications with C#, but you can also use any of the DLR languages, including IronPython. Silverlight is an exciting system. Rich interactive applications can be run in the browser, powered by the language of your choice. All of chapter 13 is devoted to Silverlight. As well as covering the basics of creating and packaging Silverlight applications, we walk through building a Silverlight Twitter client as an example.

---

[23] These features were debuted at the 2008 PDC conference by Anders Hejilsburg. See http://channel9.msdn.com/pdc2008/TL16/.

[24] Take Avalon, for example, which became WPF.

[25] Although Microsoft's idea of *cross-platform* is Mac OS X and Windows and their idea of *cross-browser* is Safari, IE, and Firefox, they've said that Opera support is in the works.

[26] See http://www.mono-project.com/Moonlight.

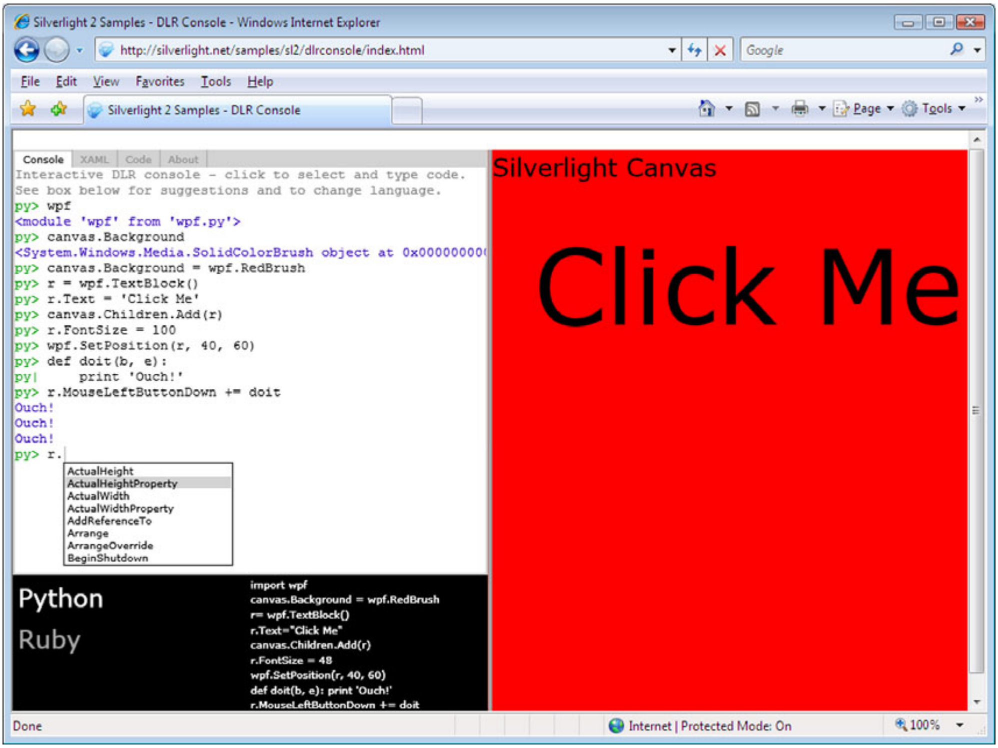


Figure 1.6 The Silverlight DLRConsole sample with a Python and Ruby Console

Figure 1.6 shows one of the Silverlight samples[27] to illustrate the dynamic power of DLR languages running in Silverlight.

This console allows you to execute code with live objects and see the results in the canvas to the right. The console supports both IronRuby and IronPython, and you can switch between them live!

Three DLR languages currently available for use with Silverlight are IronPython, IronRuby, and Managed JScript. Managed JScript is an ECMA 3–compatible implementation of JavaScript, created by Microsoft to make it easier to port AJAX applications to run on Silverlight.

Another way of using Silverlight is particularly interesting. The whole browser Document Object Model (DOM)[28] is accessible, so we can interact with and change the HTML of web pages that host Silverlight. We can also interact with normal unmanaged JavaScript: calling into Silverlight code from JavaScript and vice versa. It's already possible to write a client-side web application with the presentation layer in JavaScript, using any of the rich set of libraries available and the business logic written in IronPython.

---

[27] The DLRConsole can be downloaded as one of the samples from the Silverlight Dynamic Languages SDK at http://www.codeplex.com/sdlsdk.

[28] A system that exposes a web page as a tree of objects.

It's important to remember that, whether it runs in the browser or on the desktop, IronPython code is Python code. To understand IronPython and the place it has in the .NET world, you'll need to understand Python. It's time for an overview of the Python language.

1.2.3 The Python programming language

Python is a mature language with a thriving user community. It has traditionally been used mostly on Linux- and Unix-type platforms; but, with the predominance of Windows on the desktop, Python has also drawn quite a following in the Windows world. It's fully open source; the source code is available from the main Python website and a myriad of mirrors. Python runs on all the major platforms, plus many more, including obscure ones such as embedded controllers, Windows Mobile, and the Symbian S60 platform used in Nokia.

NOTE Python is an open source, high-level, cross-platform, dynamically typed, interpreted language.

Python was created by Guido van Rossum in 1990 while he worked for CWI in the Netherlands. It came out of his experience of creating a language called ABC, which had many features making it easy to use, but also some serious limitations. In creating Python, Guido aimed to create a new language with the good features from ABC, but without the limitations.

Python is now maintained by a core of developers, with contributions from many more individuals. Guido still leads the development and is known as the *Benevolent Dictator for Life* (BDFL), a title first bestowed during the discussions of founding a Python Software Association. Oh, and yes, Python *was* named after the Monty Python comedy crew.

Python itself is a combination of the runtime, called CPython, and a large collection of modules written in a combination of C and Python, collectively known as the standard library. The breadth and quality of the standard library has earned Python the reputation of being *batteries included*. The IronPython team has made an effort to ensure that as much of the standard library as possible still works. IronPython is doubly blessed with a full set of batteries from the .NET framework and a set of spares from the standard library.

One of the reasons for the rise in popularity of Python is the emphasis it places on clean and readable code. The greatest compliment for a Python programmer isn't that his code is clever, but that it's elegant. To get a quick overview of the guiding philosophy of Python, type `import this` at a Python console![29] You can see the result in figure 1.7.

Python is a multipurpose programming language used for all sorts of tasks. It's possible you've already used applications or tools written in Python without even being aware of it. If you've used YouTube, Yahoo Maps, or Google, then you've been using tools built or supported behind the scenes with Python.

[29] The console will need the Python standard library on its path.

```
IronPython 1.0 (1.0.61005.1977) on .NET 2.0.50727.42
Copyright (c) Microsoft Corporation. All rights reserved.
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
>>>
```

Figure 1.7 The Zen of Python, as enshrined in the Python standard library

Python quotes

Python is fast enough for our site and allows us to produce maintainable features in record times, with a minimum of developers.

—Cuong Do,
Software Architect at YouTube.com

We chose to use python because we wanted a well-supported scripting language that could extend our core code. Indeed, we wrote much more code in python than we were expecting, including all in-game screens and the main interface.

—Soren Johnson,
lead designer of Civilization IV

Python gets used for web development with web frameworks such as Zope, Django, and TurboGears. The BitTorrent application and SpamBayes (an Outlook plugin to combat spam) are both written in Python. Yum, the Linux package manager; Mnet, the distributed file store; Mailman, the GNU mailing list manager; and Trac, a popular project management and bug-tracking tool, are all written in Python. It's used by companies including Google, NASA, Sony Imageworks, Seagate, and Industrial Light & Magic. It's also used a great deal by the scientific community, particularly in bioinformatics and genomics.[30]

[30] Particularly because of the simplicity and power of Python's string handling, which is ideal for slicing and splicing gene sequences.

Because of the clarity of the syntax, Python is very easy to learn. Beginners can start with simple procedural-style programming and move into object-oriented programming as they understand the concepts, but other styles of programming are also supported by Python.

1.2.4 Multiple programming paradigms

Python is fundamentally an object-oriented[31] programming language. Everything you deal with is an object, but that doesn't mean that you're limited to the object-oriented programming style of programming in Python.

Procedural programming, the predecessor of object-oriented programming, is one alternative. Python doesn't force you to create classes if you don't want to. With the rich set of built-in datatypes and those available in the standard library, procedural programming is often appropriate for simple tasks. Because many people have past experience with procedural programming, it's common for beginners to start here. Few can avoid the allure of object-oriented programming for long, though.

Functional programming is an important programming concept. Pure functional programming allows functions to have no side effects; it can be hard to understand, but the basic concepts are straightforward enough. In functional programming languages, Python included, functions are first-class objects. You can pass functions around, and they can be used far from where they're created.

Because class and function creation is done at runtime rather than compile time, it's easy to have functions that create new functions or classes—that is, function and class factories. These make heavy use of closures. Local variables used in the same scope a function is defined in can be used from *inside* the function. They're said to have been captured by the function. This is known as *lexical scoping*.[32]

You can have parts of your code returning functions that depend on the local values where they were created. These functions can be applied to data supplied in another part of your code. Closures can also be used to populate *some* arguments of a function, but not *all* of them. A function can be wrapped inside another function with the populated arguments stored as local variables. The remaining arguments can be passed in at the point you call the *wrapper function*. This technique is called *currying*.

Metaprogramming is a style of programming that has been gaining popularity recently through languages like Python and Ruby. It's normally considered an advanced topic, but it can be useful at times. Metaprogramming techniques allow you to customize what happens at class-creation time or when attributes of objects are accessed; you can customize all attribute access, not just individual attributes (through properties).

By now we're sure you're keen to use IronPython and see what it has to offer for yourself. In the next section, we look at the interactive interpreter, and you get the

---

31 For a good introduction to object-oriented programming with Python, see http://www.voidspace.org.uk/python/articles/OOP.shtml.

32 See http://en.wikipedia.org/wiki/Lexical_scoping.

chance to try some Python code that uses .NET classes. Some basic .NET terms are explained as we go. It's fairly straightforward, but we don't give a blow-by-blow account of all the examples; explanations will come soon.

1.3 Live objects on the console: the interactive interpreter

Traditional Python (boy, does that make me feel old) is an interpreted language. The source code, in a high-level bytecode form, is evaluated and executed at runtime. Features such as dynamic object lookup and creating and modifying objects at runtime fit in well with this model. The CLR also runs bytecode, but its bytecode is optimized to work with a powerful just-in-time compiler and so .NET languages tend to be called compiled languages.

Something else that fits in well with dynamically evaluated code is an interactive interpreter, which allows you to enter and execute individual lines (and blocks) of code. By now, you should have a good overview of what IronPython is; but, to really get a feel for it, you need to use it. The interactive interpreter gives you a chance to try some Python code. We use some .NET classes directly from Python code. This is both an example of IronPython in action and a demonstration of the capabilities of the interactive interpreter.

1.3.1 Using the interactive interpreter

When you download IronPython,[33] you have two choices. You can download and install the msi installer (IronPython 2 only), which includes the Python 2.5 standard library. Alternately, you can download and unpack the binary distribution that comes as a zip file. Whichever route you take, you'll have two executables, ipy.exe and ipyw.exe, which are the equivalents of the Python executables python.exe and pythonw.exe. Both are used to launch Python scripts; ipy.exe launches them with a console window, and ipyw.exe launches programs as Windows applications (without a console).

```
ipy.exe path_to\python_script.py
```

If you run ipy.exe on its own, it starts an interactive interpreter session. The IronPython interpreter supports tab completion and coloring of the output, both of which are useful. The command line options to enable these are

```
ipy -D -X:TabCompletion -X:ColorfulConsole
```

You should see something like figure 1.8.

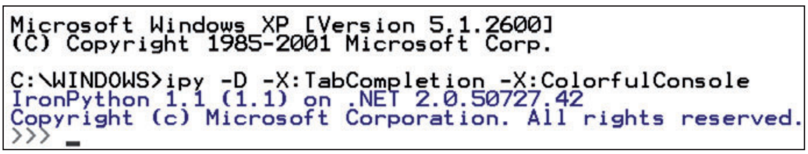


Figure 1.8 The IronPython interactive interpreter

[33] From the IronPython website on CodePlex: http://www.codeplex.com/IronPython.

IronPython is dependent on the .NET framework 2.0.[34] The interpreter, and any applications created with IronPython, will only run on computers with .NET 2.0 (or Mono) installed. In recent days, Microsoft has been pushing .NET 2.0 out to Windows machines via Windows update. A high proportion of Windows machines will already have .NET 2.0 installed. Windows machine that aren't .NET equipped will require at least the redistributable dotnetfx.exe.[35]

The interactive interpreter allows you to execute Python statements and see the results. This is known as a *'read-eval-print' loop* (REPL).[36] It can be extremely useful for exploring objects, trying out language features, or even performing quick one-off operations.

If you enter an expression, the resulting value will be printed to the console. The result of the last expression, whether value or object, is available in the interpreter by using the underscore (_).If you can't find a calculator, then you can always turn to the Python interpreter instead.

TIP Code examples to be typed into an interactive interpreter session start each line that you enter with >>> or .... This is the interpreter prompt and reflects the actual appearance of the session. It's a common convention when presenting Python examples.

```
>>> 1.2 * (64 / 2.4) + 36 +2 ** 5
100.0
>>> _
100.0
>>> x = _
>>> print x
100.0
```

More importantly, blocks of code can be entered into the interpreter, using indentation in the normal Python manner.

```
>>> def CheckNumberType(someNumber):
...   if type(someNumber) == int:
...      print 'Yup, that was an integer'
...   else:
...      numType = type(someNumber)
...      print 'Nope, not an integer: %s' % numType
...
>>> CheckNumberType(2.3)
Nope, not an integer: <type 'float'>
>>> type(CheckNumberType)
<type 'function'>
```

We're not going to get very far in this book without building an understanding of .NET terminology. Before demonstrating some more practical uses of the interpreter, we'll look at some basic .NET concepts.

---

[34] IronPython 2 requires .NET 2.0 Service Pack 1.

[35] From the memorable URL: http://www.microsoft.com/downloads/details.aspx?FamilyID=0856eacb-4362-4b0d-8edd-aab15c5e04f5&displaylang=en.

[36] From the first appearance of an interactive interpreter with the Lisp language.

1.3.2 The .NET framework: assemblies, namespaces, and references

Assemblies are compiled code (in the form of dlls or exe files)—the substance of a .NET application. Assemblies contain the classes used throughout an application.

The types in an assembly are contained in namespaces. If there are no explicitly defined namespaces, they're contained in the default namespace. Assemblies and namespaces can be spread over multiple physical files, and an assembly can contain multiple namespaces. Assemblies and namespaces are roughly the same as *packages* and *modules* in Python, but aren't directly equivalent.

For a program to use a .NET assembly, it must have a *reference* to the assembly. This must be explicitly added. To add references to .NET assemblies, you use the `clr` module. In Python, the term *module* has a different meaning than the rarely used .NET module. To use `clr`, you have to import it, and then you can call `AddReference` with the name of the assembly you want to use.

After adding a reference to the assembly, you're then free to import names[37] from namespaces it contains into your IronPython program and use them.

```
>>> import clr
>>> clr.AddReference('System.Drawing')
>>> from System.Drawing import Color, Point
>>> Point(10, 30)
<System.Drawing.Point object at 0x0...2B [{X=10,Y=30}]>
>>> Color.Red
<System.Drawing.Color object at 0x0...2C [Color [Red]]>
```

There are a few exceptions. The IronPython engine already has a reference to the assemblies it uses, such as `System` and `mscorlib`. There's no need to add explicit references to these.

The `clr` module includes more goodies. It has different ways of adding references to assemblies, including specifying precisely which version you require. We take a more detailed look at some of these later in the book. For now, we demonstrate how to use live objects from the interactive interpreter.

1.3.3 Live objects and the interactive interpreter

The interactive interpreter is shown off at its best when it's used with live classes. To illustrate this, here's some example code using Windows Forms. It uses the `System.Windows.Forms` and `System.Drawing` assemblies.

```
>>> import clr
>>> clr.AddReference('System.Windows.Forms')
>>> clr.AddReference('System.Drawing')
>>> from System.Windows.Forms import Application, Button, Form
>>> from System.Drawing import Point
>>> x = 0
>>> y = 0
>>> form = Form()
>>> form.Text = "Hello World"
```

Imports names

Instantiates form

[37] The imported names are names that refer to objects. These will usually be classes when importing from a .NET namespace. You'll learn more about Python imports in chapter 2.

So now you should've created a form with the title *Hello World*. Because you haven't yet started the application loop, there's no guarantee that it will be visible. Even if it is visible, it will be unresponsive, as you can see in figure 1.9.

```
>>> button = Button(Text="Button Text")
>>> form.Controls.Add(button)
>>> def click(sender, event):
...     global x, y
...     button.Location = Point(x, y)
...     x += 5
...     y += 5
...
>>> button.Click += click
>>> Application.Run(form)
```

Adds button to form (lines 1–2)
Defines click handler function (def click … y += 5)
Starts application loop (last two lines)



Figure 1.9 A Hello World form, shown before the event loop is started

When the application event loop is started and the form is shown, the form will look like figure 1.10. Every time you click the button, it will move diagonally across the form.

We cover all the details of what is going on here later; the important thing to note is that the whole process was done live.

If you run this demonstration, you'll notice that when you execute the `Application.Run(form)` command, the console loses control. Control doesn't return to the console until you exit the form because starting the application loop takes over the thread it happens on.



Figure 1.10 Active Hello World form with a button

The IronPython winforms sample

Various pieces of sample code are available for IronPython. One of these is a useful piece of code called `winforms`.

If you run the IronPython console from the tutorial directory and `import winforms`, then the console is set up on another thread, and commands you enter are invoked back onto the GUI thread. You can create live GUI objects, even though the application loop has started, and see the results immediately.

With `winforms` imported, you can display the form by calling `form.Show()`. The event loop has already been started, so there's no need to call `Application.Run`.

As well as enabling you to work with live objects, Python has powerful introspective capabilities. A couple of convenience commands for using introspection are particularly effective in the interpreter.

1.3.4 Object introspection with dir and help

It's possible when programming to occasionally forget what properties and methods are available on an object. The interpreter is a great place to try things out, and two commands are particularly useful.

`dir(object)` will give you a list of all the attributes available on an object. It returns a list of strings, so you can filter it or do anything else you might do with a list. The following code snippet looks for all the interfaces in the `System.Collections` namespace by building a list and then printing all the members whose name begins with *I*.

```
>>> import System.Collections
>>> interfaces = [entry for entry in dir(System.Collections)
... if entry.startswith('I')]
>>> for entry in interfaces:
...   print entry
...
ICollection
IComparer
IDictionary
IDictionaryEnumerator
IEnumerable
IEnumerator
IEqualityComparer
IHashCodeProvider
IList
>>>
```

The next command is `help`. `help` is a built-in function for providing information on objects and methods. It can tell you the arguments that methods take; for .NET methods, it can tell you what types of objects the arguments need to be. Sometimes the result contains other useful information.

As an example, let's use `help` to look at the `Push` method of `System.Collections.Stack`.

```
>>> from System.Collections import Stack
>>> help(Stack.Push)
Help on method-descriptor Push
 |  Push(...)
 |      Push(self, object obj)
 |
 |        Inserts an object at the top of the
 |         System.Collections.Stack.
 |
 |        obj: The System.Object to push onto the
 |         System.Collections.Stack. The value can be null.
```

`help` will also display docstrings defined on Python objects. *Docstrings* are strings that appear inline with modules, functions, classes, and methods to explain the purpose of the object. Many members of the Python standard library have useful docstrings

defined. If you use `help` on the `makedirs` function from the `os` module, you get the following:

```
>>> import os
>>> help(os.makedirs)
Help on function makedirs in module os
 | makedirs(name, mode)
 |    makedirs(path [, mode=0777])
 |
 |       Super-mkdir; create a leaf directory and all intermediate ones.
 |       Works like mkdir, except that any intermediate path segment (not
 |       just the rightmost) will be created if it does not exist.  This is
 |       recursive.
 |
```

The interactive interpreter is an extremely useful environment for exploring .NET assemblies. You can import objects and construct them live to explore how they work. It's also useful for one-off jobs such as transforming data. You can pull the data in, push it back out again, and walk away.[38]

1.4 Summary

You've seen that Python is a flexible and powerful language suitable to a range of tasks. IronPython is a faithful implementation of Python for the .NET platform, which itself is no slouch when it comes to power and features. Whether you're interested in writing full applications, tackling scripting tasks, or embedding a scripting language into another application, IronPython has something to offer you.

Through the course of this book, we demonstrate these different practical uses of IronPython. There are also sections that provide reference matter for the hard work of turning ideas into reality when it comes to real code.

The interactive interpreter is important for experimentation. It's a great tool for trying things out, reminding you of syntax or language features, and for examining the properties of objects. The `dir` and `help` commands illustrate the ease of introspection with Python, and we expect that, as you work through more complex examples, they'll be helpful companions on the journey.

In the last section, you got your feet wet with Python; but, before you can do anything useful with it, you'll need to learn a bit more of the language. The next chapter is a fast-paced tutorial that will lay the foundations for the examples built in the rest of the book.

[38] This is a quote from Python expert Tim Golden who does a lot of work with databases and Python, some of it using only the interactive interpreter.

Introduction to Python

This chapter covers

- The Python datatypes
- Basic language constructs
- Exception handling, list comprehensions, and closures
- Python project structure: modules and packages
- The Python standard library

Programming is a craft. Knowing the rules of a programming language is the science behind the art of creating well-structured and elegant programs. Your programs may be small and functional, or they may be vast cathedrals adorned with the intricate masonry of recursive algorithms and data structures. The joy of programming is found in the process of creation as much as in the final result.

In the first chapter, you had a taste of Python, which is essential to programming with IronPython; luckily, Python is easy to learn. This chapter is a fast-paced Python tutorial that will give you a great foundation in Python the language, both for following the examples in this book and for creating your own applications.

This chapter does assume some experience in programming, but not in any particular language, so it should be easy to follow. Even if you're already fluent in Python, you should skim this chapter because there are interesting details that are

specific to IronPython.[1] If you're not already a fluent Python speaker, the goal of this chapter is for you to learn enough Python so that you can confidently tackle creating a well-structured IronPython application in the coming chapters.

Learning a programming language is itself like a work of construction—say, building a house. The basic datatypes, like mortar, bind programs together, and they get everywhere! Next come the larger elements—the bricks and blocks and supporting beams; in Python, these are the functions, classes, and modules that give your program shape and structure. You need to understand these things before moving on to the wiring, the floorboards, and the thousand-and-one other details. (It's no wonder they call it software architecture!) This order is roughly the one that the tutorial will follow. We won't get to all the decorations and furnishings; we'll leave you with plenty more to learn.

Before you dive in, it's a good idea to have a good programmer's editor on hand, such as a text editor or an IDE that supports Python.

A proper IDE will give you additional features such as autocomplete, source code navigation, smart indent, project browsing, and a whole lot more. Some programmers get by without these features and stick to using a text editor (my boss included), but I (Michael) find that the tools provided by a good IDE are invaluable. My personal favorite IDE for Python is a commercial one called Wing,[2] shown in figure 2.1. The

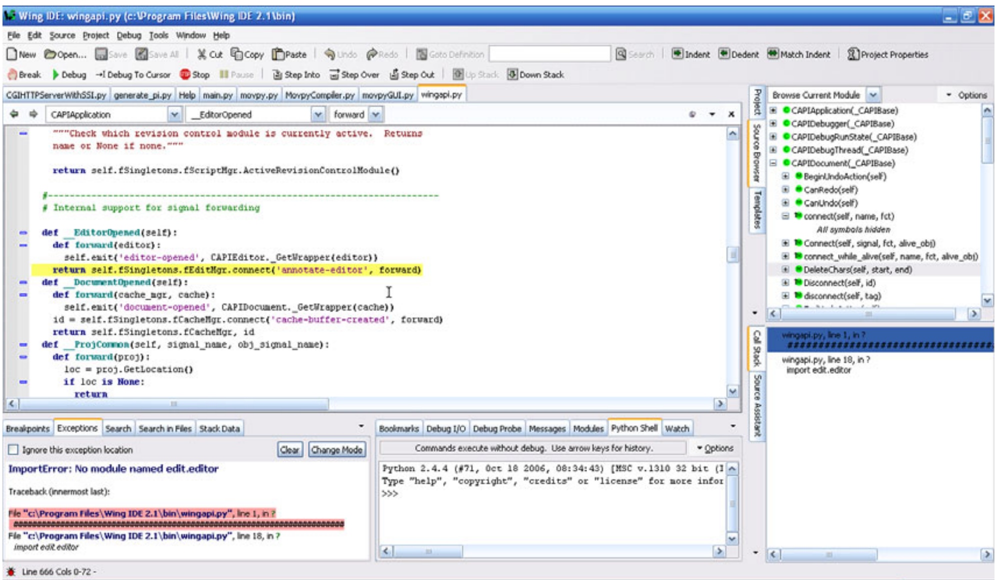


Figure 2.1 Wing IDE, with its built-in project browser and interactive interpreter, is a great IDE for Python.

---

[1] We've highlighted particular differences between Python and IronPython in callouts and sidebars.

[2] See http://www.wingware.com.

autocomplete is the most accurate[3] I've seen in a Python editor. Plenty of other good Python IDEs, both free and commercial, are available.

2.1 An overview of Python

As we mentioned earlier, Python supports several different programming techniques. For simple tasks, which there may be many of in a complex program, functions can be more appropriate than making everything a class. Functional programming can make it hard for others to follow your code if it's overused, but it can also be extremely powerful. With Python, you're free to mix procedural, functional, and object-oriented programming to your heart's delight; and, because they all have their place, you probably will. First things first, though—to do this, you need to know the language.

The core object model of Python is similar to imperative languages such as C#, Java, and VB.NET.[4] If you've used any of these languages before, even if aspects of the syntax are unfamiliar, you'll find learning Python easy. Python does differ from these languages in how it delimits blocks of code. Instead of using curly braces, Python uses indentation to mark blocks of code. Here's a simple example using an `if` block:

```
if condition == True:
   do_something()
```

The statements to be executed if the condition is `True` are indented relative to the `if` statement. You can use any indentation you want (tabs or two spaces or four spaces) as long as you're consistent (and *never* mix tabs and spaces in the same file). Because we use a lot of Python code in this chapter, you'll be seeing a lot more of Python's indentation rules.

NOTE Two simple things to know about Python: Python is case sensitive, and the comment symbol is `#`.

In object-oriented programming languages, everything is an object. Every object has a type, and the simplest (conceptually) of these are the ones built into Python, such as strings and numbers. These are the basic datatypes; and, as the mortar of the house we're constructing, we look at them first. Different kinds of Python objects are shown in figure 2.2 and are arranged in layers from the most simple to the more complex.

In terms of the basic datatypes, the creators of IronPython have gone to great lengths to ensure that IronPython is a faithful implementation of Python. The structure you learn about here is just as true for IronPython as it is for the regular Python distribution, CPython.

---

[3] Python is a highly dynamic language and so the type of some objects can't be inferred; the type is determined only at runtime, making it hard for IDEs to always know what attributes are available. Several IDEs do an excellent job, but Wing is the best of the ones I've tried.

[4] Whereas languages like Ruby inherit more from Smalltalk in their core object model.

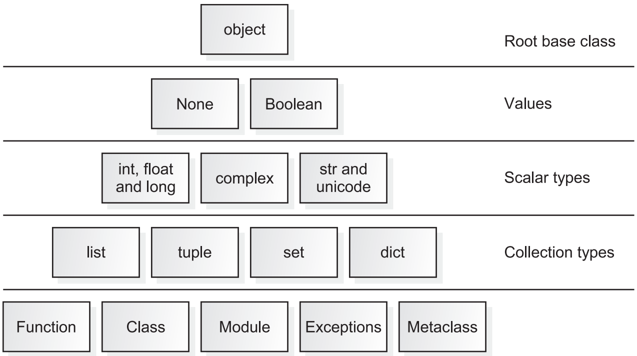


Figure 2.2 This Python object pyramid shows some of the Python types. The layers are arranged in approximate order of complexity (from top to bottom).

2.1.1 Python datatypes

Python boasts a rich set of built-in datatypes that you can use without importing any modules. Most of these, with the exception of the set that we introduce shortly, have syntax for including them within your code. To understand the examples in this book or to write Python yourself, you're going to need to recognize them. Table 2.1 contains a guide to all the basic datatypes and their syntaxes.

Table 2.1 The built-in datatypes and their syntaxes

Datatype	Type	Syntax examples	Notes

Byte string	`str`	`'hello world'`
		`"also hello world"`
		`"""A triple quoted multiline string"""`
		`'''Another triple quoted multiline string'''`
			In IronPython, byte strings and Unicode strings are the same datatype.

Unicode string	`unicode`	`u'A Unicode string'`
		`u"Another Unicode string"`
		`u"""Yet another Unicode string."""`	

Integer	`int`	`3` `-3`	
Long integer	`long`	`9L` `9999999999999999999999L`	

Floating point	`float`	`0.0`
		`-3.1`
		`2e100`
		`2.765e-10`	

Complex numbers	`complex`	`2 + 3j` `8j`	

Table 2.1 The built-in datatypes and their syntaxes *(continued)*

Datatype	Type	Syntax examples	Notes
List	`list`	`[]`	An empty list.
		`[1, 2, 3, 4]`	A populated list.
		`[1, "a", 3.0]`	A list containing members of different types.
Tuple	`tuple`	`()`	An empty tuple.
		`(1,)`	Tuple with a single member.[a]
		`(1, 2, 3, 4)`	
		`1, 2, 3, 4`	Although tuples are usually created with parentheses, they're only needed to disambiguate in situations where commas have other significance (such as argument lists in function calls). It's the commas that are significant in creating a tuple.
Dictionary	`dict`	`{}`	An empty dictionary.
		`{'key': 'value', 'key2': 'value2'}`	A populated dictionary.
Set	`set`	`set()`	Creates an empty set.
		`set([1, 2, 3, 4])`	Creates a set from a list.
None	`NoneType`	`None`	Equivalent of `NULL` in other languages.
Boolean	`bool`	`True`	
		`False`	

a. That trailing comma is important! In the case of a tuple with only one member, it's needed to disambiguate a tuple from an ordinary expression surrounded by parentheses.

This table isn't exhaustive. For example, it doesn't include things like the slightly exotic `frozenset` or the built-in exceptions.

In addition to the information in the table, you need to know some additional things about these datatypes.

STRINGS

You include Python string literals in your code with quotes. You can use single quotes or double quotes—there's no difference. If you want to include literals with newline characters, you can use triple-quoted strings.

You can also use normal Unix-type escape characters.[5] For example, here's a string with a hard tab separating the two words and terminated with a newline character:

```
'hello\tworld\n'
```

[5] See this page for a full list of the escape characters: http://docs.python.org/ref/strings.html.

The standard string type is often referred to as the byte string; the contents are stored internally as a sequence of bytes representing the characters. Byte strings can also be used for storing binary data.

Python has a second string type, the Unicode string. You create a Unicode string literal by prefixing your string with a *u*.

```
u'This is a Unicode string'
```

NOTE Because IronPython is built on top of .NET, the Unicode and string types are the same type. This is better than having two string types (and is the position that Python will be in by Python 3.0), and you should have less encoding-related problems with IronPython than you would with CPython.

Both `str` and `unicode` strings have a common base class: `basestring`. If you want to check if an object is a string and you don't care which type of string, you can use `isinstance(someObject, basestring)`. This isn't important for IronPython, but can be useful for compatibility with CPython.

You may stumble across a few other syntax permutations for strings. For example, by prefixing a string with an *r*, you make it a raw string. Backslashes aren't interpreted as escape characters in raw strings.[6] Raw strings are especially useful for regular expressions where backslashes have syntactic meaning and having to continually escape them makes regular expressions less readable.

All the built-in types (except for `None` and tuples) have a large number of methods that do useful things. Strings are no exception; they have methods for trimming whitespace, joining strings, checking if they contain substrings, and much more. For a list of the Python methods available on strings, visit http://docs.python.org/lib/string-methods.html.

If you're a .NET developer, then you'll already be familiar with the .NET methods available on strings.

THE CLR MODULE AND IRONPYTHON TYPES

One thing that makes working with IronPython so straightforward is that the basic datatypes are both Python objects and .NET objects. An IronPython string is also a .NET string—which is clever magic. By default the .NET methods aren't available on the basic types such as strings.

```
>>> string = 'Hello World'
>>> string.ToUpper()
Traceback (most recent call last):
AttributeError: 'str' object has no attribute 'ToUpper'
```

The IronPython developers made this decision after much debate; in the end, they felt that leaving extra attributes on Python objects wouldn't be a faithful implementation of Python. You can enable the .NET methods by importing the `clr` module.

[6] With the odd exception that a raw string can't end in an odd number of backslashes; otherwise, the last backslash escapes the closing quote.

```
>>> import clr
>>> string.ToUpper()
'HELLO WORLD'
```

NOTE The `clr` module provides functions for interacting with the underlying .NET runtime. It's the basic entry point for .NET/IronPython interoperation, and we use it a great deal throughout the book.

You can easily determine that IronPython strings are also .NET strings. If you compare the Python `str` type with the .NET `System.String` type, you'll see that they are, in fact, the same object.

```
>>> import System
>>> System.String
<type 'str'>
>>> System.String is str
True
```

When using the basic types, you can choose whether to use .NET patterns or Python patterns of programming from within IronPython code. As we go through the book, you'll find that we also have this choice when dealing with higher level concepts such as working with files or threading.

We've looked at strings; now let's look at the next most common objects—numbers.

NUMBERS

Python has four types for representing numbers: integers, long integers, floating point numbers, and complex numbers.

Integers are for representing whole numbers, positive or negative, up to `sys.maxint`,[7] which is a value that depends on the underlying platform. Long integers are used to represent integers that are greater than this number. Internally, Python promotes large numbers into longs automatically; in practice, you'll rarely care whether a number is an int or a long. With Python long integers, the maximum number you can represent is only limited by the amount of memory you have. That's likely to be quite a big number.

You can mix operations involving integers and floats; the result will always be a float.

If you divide two integers, you'll get an integer back. This can be surprising if you aren't expecting it. If you need to get the real value of a division back, then make sure that one of the values is a float.

```
>>> 9/2
4
>>> 9/2.0
4.5
```

Python also has syntax for complex numbers, which are unusual, but handy for certain kinds of math. Having said that, I don't think I've ever needed to use them.

[7] Typically on a 32-bit operating system, this will be the largest number that can be stored in a 32-bit signed integer: 2147483647.

Integer division

Dividing two integers in Python returns an integer. You can change this by enabling true division, which will be the default behavior in Python 3.0. True division is a `__future__` option; you enable it by having the following import as the first statement in a Python file:

```
from __future__ import division
```

Because `__future__` is a standard library module, it depends on having the Python standard library on your path.

If you want truncating division, you should use the `//` floor division operator, which behaves the same whether or not true division is enabled. (If one of the operands is a float, then the result will still be a float, but truncated down to the integer value.)

Numbers and strings are fine for representing individual values, but often you need to store more complicated information, which can involve creating nested data structures. To do this, you need datatypes capable of containing values—the container datatypes. The first of these is the list.

LISTS

The list is a very flexible datatype. You can store objects of any type in lists, and they grow themselves in memory—you don't need to specify a size for them. You can use them with the minimum of boilerplate code.

NOTE As the name implies, lists are used for storing multiple objects. The objects are stored in a sequence, one after another, and you access them by their positions in the list. In Python terminology, the objects contained in a list (or other containers) are usually referred to as *members*. This is slightly different than the way .NET terminology uses *member*, which corresponds more closely to the Python term *attribute*.

The list is the archetypal sequence type in Python. You access the members of a list by position, a process called *indexing*; the first member is at position zero. You can also access members starting from the end, by using negative numbers.

```
>>> a = [1, 2, 3, 4]
>>> a[0]
1
>>> a[-1]
4
```

Attempting to access a member outside the bounds of the list will raise an `IndexError`.

```
>>> a = [0, 1, 2, 3]
>>> a[4]
Traceback (most recent call last):
IndexError: index out of range: 4
>>> a[-5]
Traceback (most recent call last):
IndexError: index out of range: -5
```

Sequences and iterables

Sequences are examples of a type of object that you can iterate over. You can also create other types of objects that can be iterated over: iterators and generators.

A key fact about sequences is that you can index them, or ask them for their nth member. Because iterators and generators produce their members dynamically, you can't index into them.

In many places, it won't matter whether you pass in a sequence or an iterator. In that case, a function or method may just specify that it needs an iterable.

Assigning to members of a list uses the same syntax.

```
>>> a = [1, 2, 3, 4]
>>> a[0] = 'a'
>>> a
['a', 2, 3, 4]
>>> a[-1] = 'b'
>>> a
['a', 2, 3, 'b']
```

Deleting members uses the `del` keyword.

```
>>> del a[0]
```

Because you can modify a list by changing members or adding and removing members, it is a mutable datatype. The list has a close cousin that's immutable: the tuple.

TUPLES

The tuple is a container object similar to a list. Tuples can be indexed in the same way as lists; but, because they're immutable, you can't assign to members. Once a tuple has been created, you can't add or remove members. Because of this restriction, tuples are more efficient (both in memory use and speed) than lists. The fact that they're immutable also means that they can be used as dictionary keys—which is handy.

By convention in Python, tuples are used when the elements are heterogeneous (different types of objects); lists are used for homogeneous collections. This is a distinction observed more in theory than in practice.

Tuples often appear implicitly in Python. For example, when a function returns multiple values, they're returned as a tuple.

Both tuples and lists are container types that provide access to their members by position (sequences). To check to see if a sequence contains a particular member, the interpreter has to search it. If the sequence doesn't contain the object you're looking for, every member will have to be checked, a very inefficient way of searching. Other container types provide a more efficient way of checking membership. One of these is the dictionary, which stores values associated with a key.

DICTIONARIES

The dictionary is the archetypal mapping type in Python. (The actual type is `dict`.) Dictionaries store values mapped to keys, which can be any immutable value.[8] Dictionaries with strings as keys map names to objects; much of Python is implemented on top of dictionaries.

Fetching a value from a dictionary is done using the same syntax as indexing a list, but using the key rather than an index position.

```
>>> x = {'Key 1': 'Value number 1',
... 'Key 2': 'Value number 2'}
>>> x['Key 1']
'Value number 1'
>>> x['Key 2']
'Value number 2'
```

If you attempt to access a dictionary with a key that doesn't exist, then a `KeyError` will be raised.

```
>>> x = {'Key 1': 'Value number 1',
... 'Key 2': 'Value number 2'}
>>> x['Key 3']
Traceback (most recent call last):
KeyError: 'Key 3'
```

Adding new members to a dictionary, changing existing ones, or deleting entries is trivially easy.

```
>>> a = {}
>>> a[(0, 1)] = 'Get the point?'
>>> a
{(0, 1): 'Get the point?'}
>>> a[(0, 1)] = 'something else'
>>> a
{(0, 1): 'something else'}
>>> del a[(0, 1)]
>>> a
{}
```

You probably won't be astonished to learn that dictionaries have a host of useful methods. You can find a list of them at http://docs.python.org/lib/typesmapping.html.

If you need to store groups of data where you know each member will be unique, sets can be more appropriate than a dictionary. Sets don't require you to create a key for the objects stored inside them.

SETS

Unlike the other types we've looked at so far, there's no built-in syntax for creating sets. Instead you call the set constructor, passing in an iterable like a list or a tuple. Sets are used for containing members, where each member is unique. Checking to see if a member is contained in a set is much faster than checking for membership of a list.

8 Well, technically any hashable value. This allows you to implement custom objects and control how or whether they behave as dictionary keys. This is a subject for another page, though.

```
>>> x = set([1, 2, 3, 4])
>>> 5 in x
False
>>> 2 in x
True
```

Sets are iterable, but they're inherently unordered; even if the order of iteration *appears* to be consistent, you shouldn't rely on it in your code. Sets have methods for adding members; calculating the union, intersection, and difference between sets; and much more. See http://docs.python.org/lib/set-objects.html.

You've now met all the basic datatypes that carry values. Two more types, which you'll use to represent values or information in your programs, have fixed values. These types are `None` and the Booleans.[9]

NONE AND THE BOOLEANS

`None`, `True`, and `False` are useful values that turn up all the time.

`None` represents null values and, like tuples, sometimes turns up implicitly in Python. If a function has no explicit return value, then `None` is returned.

`True` and `False` are the Boolean values, and are returned by comparison operations.

```
>>> 1 == 2
False
>>> 3 < 4
True
```

In Boolean terms, the following objects are considered to be `False`:

- `False` and `None`
- 0 (integer, long, or float)
- An empty string (byte-string or Unicode)
- An empty list, tuple, set, or dictionary

Everything else is considered `True`.

```
>>> x = {}
>>> if not x:
...   print 'Not this one'
...
Not this one
>>> y = [1]
>>> if y:
...    print 'This one'
...
This one
```

We've now covered the fundamental built-in types. You should be able to recognize these when they appear in Python source code and understand operations performed on them. You also know how to include these types in your own programs.

To be able to use them effectively, you need to understand how Python treats variables, the names you use to refer to your objects.

[9] Which sounds like a great name for a band…

2.1.2 Names, objects, and references

Although Python supports several different programming paradigms, it's built on solid object-oriented principles. A natural consequence of this is that everything in Python is an object.

NOTE All the built-in types inherit from the Python class `object`. All IronPython objects inherit from the .NET `System.Object`.

If you've programmed in C#, you'll be used to the difference between value types and reference types. When you pass a value type, you're passing the value (the data) itself. When you pass a reference type, you're passing a reference to the object rather than the object itself. From this reference, you can access the properties of the object, call its methods, and so on.

With Python, there are *only* reference types. This rule is straightforward, but it does have some consequences that can be surprising for programmers new to Python.

Python draws an important distinction between objects and the names you use to refer to them. When you assign a value to a variable, you're creating a name bound to an object. If you later assign a new value to the variable, you're removing the reference to the original object and rebinding the name to a new object.

```
>>> a = 3      ❶
>>> b = a        ❷
>>> a = 6      ❸
>>> b        ❹
3
```

This is illustrated in figure 2.3.

The effect of ❷, shown in ❹, can confuse new programmers. The name *b* is bound to the object pointed to by the name *a* ❷ (which is an integer with the value of three ❶). New programmers might expect the line `b = a` to cause the variable b to have the same value as the variable a, so that when a is changed, b ought to change as well. Instead, when `a = 6` is executed, the name *a* is rebound to point to a different object ❸, whereas the name *b* remains bound to the original one ❹.

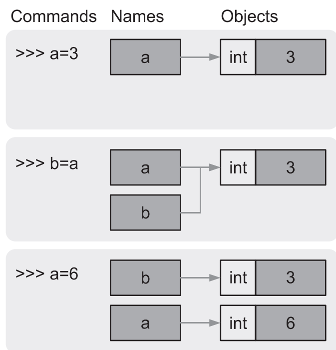


Figure 2.3 Names bound to objects by assignment statements

A more succinct way of saying this is that Python doesn't really have variables. Python has objects and names that reference objects. Understanding this is central to understanding Python. If you already understand the difference between reference and value types, then this is probably pretty intuitive to you. If you don't feel you understand it yet, don't worry about it for now; it should become clearer as you use Python more.

Although Python doesn't have the reference type/value type distinction, it does draw a distinction between mutable and immutable datatypes.

2.1.3 Mutable and immutable objects

You've already seen that reassigning a name points it to a new object. Often you want to change or update the value a name points at; this might be incrementally building a string, adding members to a list, or updating a counter.

Some objects can be changed in place without having to create a new object. These are mutable objects. Others can't be changed once they've been created. These are immutable objects.

As you might expect, the container types `list` and `dict` (dictionaries) are mutable. Adding new members doesn't create a new object, but modifies the object in place. Tuples are immutable. Attempting to change or delete members, or add new ones, will raise an exception; but, tuples can contain mutable objects.

Strings and the number types are also immutable. Operations that change values don't change the underlying object but create new ones.

```
>>> a = b = 'a string'
>>> a += ' which is getting bigger'
>>> a
'a string which is getting bigger'
>>> b
'a string'
```

Here the in-place operator `+=` is merely syntactic sugar[10] for `a = a + 'which is getting bigger'`. The name *a* is re-bound to a new string which is formed by adding `'a string'` to `'which is getting bigger'`. The name *b* remains bound to the string `'a string'`.

The last two sections might seem confusing at first; but, once you've grasped this, you've understood a lot about Python. Here's a simple test:

```
>>> a = []
>>> b = [a]
>>> a.append(1)
```

Can you predict what the object pointed to by the name *b* contains?[11]

At last you've reached the end of the section devoted to the basic datatypes. It's time to move from the mortar to the bricks. Functions and classes contain code, which uses the basic types. This code will be comprised of statements and expressions, so the first part of the section on basic constructs will cover these.

2.2 Python: basic constructs

We've now covered the basic datatypes and the way that Python handles names and objects. These things are important, but objects alone do not a program make. You also need to be able to do things with these objects. Constructs common to many programming languages, such as functions and classes and loops and conditionals, are

---

[10] *Syntactic sugar* means providing a nicer syntax for something that's possible another way.

[11] The answer is `[[1]]`. This list pointed to by the name *b* contains the list pointed to by the name *a*, which now has a member.

needed to form the substance of programs. This section is about the different constructs available to you in Python.

The lowest level of constructs consists of statements and expressions. Every line of Python code will have at least one statement or expression. Statements and expressions are the raw ingredients out of which you create conditionals and loops. Stepping up a level, from these you'll craft functions and classes and, up the final step to full applications, organize functions and classes into modules and packages.

2.2.1 Statements and expressions

Python draws a distinction between statements and expressions. This wouldn't be true in a purely functional programming language where everything is an expression.

NOTE A statement does something, performs an action, whereas an expression returns a value.

Statements are actions such as printing, assigning a value, or raising an exception. A statement doesn't return a value, so you can't get away with the following:

```
>>> x = print 3
Traceback (most recent call last):
SyntaxError: unexpected token print (<stdin>, line 1)
```

Statements can't be compounded together. You can do multiple assignments as a single statement.

```
>>> x = y = 10
```

Statements other than assignment use a Python keyword. These keywords are reserved words, and you can't use them as variable names. You can see the full list of reserved keywords (not all of which form statements) at http://docs.python.org/ref/keywords.html.

Python follows the normal BODMAS[12] precedence rules for operators in expressions. The best reference I could find on Python operators is from *A Byte of Python* by CH Swaroop at http://www.swaroopch.com/notes/Python_en:Operators_and_Expressions#Operator_Precedence.

Parentheses not only group parts of an expression together, but also allow you to break unwieldy expressions over multiple lines.

```
>>> (1 + 2 + 3 + 4 + 5 + 6 + 7 +
... 8 + 9 + 10)
55
```

Python uses keywords and syntax for its most basic elements that will be familiar to most programmers, and are easy to understand even if they aren't familiar. The most fundamental of these are conditionals and loops.

[12] Brackets, Orders, Division and Multiplication, Addition and Subtraction. See http://www.mathsisfun.com/operation-order-bodmas.html.

2.2.2 Conditionals and loops

Very often through the course of a program you'll need it to take different actions depending on some condition—maybe finding the terminating condition of an algorithm, or responding to user input. The conditional statement in Python uses the `if` keyword.

Every statement that starts a new block of code must be terminated with a colon. Indented code following the `if` statement will only be executed if the condition evaluates to `True`. As soon as your code is dedented, execution continues normally. `if` allows you to provide multiple branches for different conditions; a final branch will be executed if none of the other conditions have been met. `elif` is used for multiple branches, and `else` for the final branch.

```
>>> x = -9
>>> if x == 0:
...     print 'x equals zero'
... elif x > 0:
...     print 'x is positive'
... else:
...     print 'x is negative'
...
'x is negative'
```

Another conditional in Python is `while`, which is also a loop. The `while` loop repeats a block of code until a condition is met, or until you explicitly break out of the loop.

```
>>> a = 0
>>> while a < 2:
...     print a
...     a += 1
...
0
1
```

If you want to terminate the loop early, you can use the `break` statement. With an optional `else` clause, you can tell whether a loop completed normally.

```
>>> a = 0
>>> while a < 6:
...    print a
...    if a >= 2:
...       break
... else:
...     print 'Loop terminated normally'
0
1
2
```

The second kind of loop in Python is the `for` loop. It's used to iterate over a sequence (or any iterable), and allows you to perform an operation on every member.

```
>>> sequence = [1, 2, 3]
>>> for entry in sequence:
...     print entry
...
```

```
1
2
3
```

Although it's theoretically possible to write a program using only the constructs we've examined so far, it would be an extremely ugly program. To reuse blocks of code (and write well-structured, modular programs), you need functions and classes.

2.2.3 Functions

Functions allow you to break code down into smaller units and make programs more readable. Functions take values as arguments and return values, but they may also be used for their side effects, such as writing to the filesystem or a database.

Like other aspects of Python, the syntax for functions is straightforward. It's shown in figure 2.4.

A function is created using the `def` keyword, followed by the function name and an argument list (which can be empty if the function receives no arguments). Because they introduce a new block of code, function definitions are terminated with a colon.

The function body is the indented block which follows the `def`. If the interpreter hits a `return` statement, then it leaves the function and returns the specified value (or `None` if no value is supplied). If the interpreter reaches the end of the function body without hitting a return statement at all, then `None` is also returned.

If the function needs to receive arguments, then a comma-separated list of names is placed between the parentheses that follow the function name. When the function is called, a value must be supplied for every name in the argument list.

Python also allows you to define function arguments that have a default value. If the argument isn't supplied when the function is called, the default value is used instead. This is done by assigning a value to the argument inside the function definition.

```
>>> def PrintArgs(arg1, arg2=3):
...     print arg1, arg2
...
>>> PrintArgs(3)
3 3
>>> PrintArgs(3, 2)
3 2
```

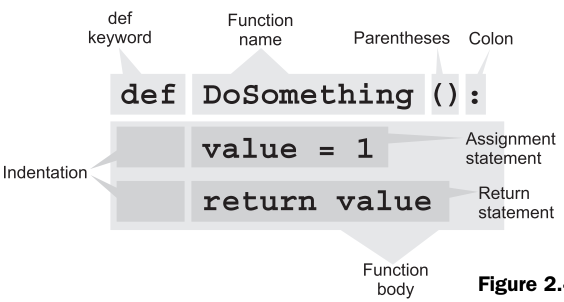


Figure 2.4 Function definition, body, and return

As you can see, when you don't pass in an explicit second value, `arg2` takes on the default value specified in the function definition. If you do pass in a second value, it's used instead of the default one. Arguments with default values are known as *keyword arguments*. Note that you can also pass in keyword arguments by position; this is shown in the last line of the following example.

```
>>> def ShowingOffKeywords(a=None, b=None):
...     if a is not None:
...        print 'a =', a
...     if b is not None:
...        print 'b =', b
...
>>> ShowingOffKeywords(a=3)
a = 3
>>> ShowingOffKeywords(b='a string')
b = a string
>>> ShowingOffKeywords(2.0, 3)
a = 2.0
b = 3
```

If used sparingly, keyword arguments can greatly improve an API by removing the need to pass in values that rarely differ from the defaults. If you overuse them, you can create a complex API that no one will remember how to use. Keyword arguments also provide a convenient mechanism for configuring .NET classes—the ultimate goal of this tutorial.

NOTE Callable objects can be called with arguments and return values. Functions are callable objects, but they're not the only things that are callable. This is why you might see the term *callable* used instead of *function* in places through this book.

Functions are great for simple and self-contained operations where the overhead of writing a class isn't needed. Many simple operations end up being needed in program after program. Instead of having to re-implement your own functions over and over again, Python has a selection of built-in functions.

2.2.4 Built-in functions

There are many operations that you need to perform often or that can be awkward to do cleanly. For a lot of these, Python provides built-in functions that are always available. Unlike the reserved keywords, you *can* create names (variables or functions) that shadow these names. Needless to say, shadowing built-ins is bad practice.

Table 2.2 shows some of the most useful built-in functions, along with a brief explanation.

Many of these functions have optional arguments or alternative usage patterns. For details, refer to the Python documentation, which provides a list of all the built-in functions at http://docs.python.org/lib/built-in-funcs.html.

Table 2.2 Some commonly used built-in functions

Name	Example	Purpose
`abs`	`abs(-3)`	Returns the absolute value (positive) of a number passed in.
`enumerate`	`enumerate(['a', 'b', 'c'])`	Returns an iterable (an `enumerate` object). Iterating over this returns a sequence of tuples: `(index, element)`. Index is the position of the element in the sequence or iterable you pass in.
`len`	`len([1, 2, 3])`	Returns the length (number of elements) of an object passed in (sequence, iterable, set, or dictionary and friends).
`isinstance`	`isinstance(myObject, SomeType)`	Tells you if an object is an instance of a specified type, or one of its subclasses.
`max`	`max(6, 10, 12, 8)`	Returns the largest of all the arguments you pass in, or the largest member of a sequence or iterable.
	`max([1, 2, 3, 4])`	
`min`	`min(6, 10, 12, 8)`	Like `max`, except it returns the smallest instead of the largest.
	`min([1, 2, 3, 4])`	
`open`	`open(filename, mode)`	Opens the specified file with the specified mode.[a]
`range`	`range(10)`	Returns a list with elements from 0 up to (but not including) the number you pass in.
	`range(5, 10)`	Returns a list with elements from the first value you pass in, up to (but not including) the second value you pass in.
`reversed`	`reversed(sequence)`	Returns a sequence, which is a reversed version of the one you pass in. (It does *not* alter the sequence that you pass in.)
`sorted`	`sorted(iterable)`	Returns a *new* list, which is a sorted version of the sequence or iterable that you pass in.
`sum`	`sum(iterable)`	Returns the sum of all the members of the sequence or iterable you pass in.
`zip`	`zip(iterable1, iterable2...)`	Returns a list of tuples, with corresponding members from each of the sequences or iterables that you pass in. The first tuple will have the first member from each argument, the second tuple will have the second member, and so on. The list returned is only as long as the shortest one that you pass in.

a. The filename can be a path relative to the current directory or an absolute path. The mode should be a string. See the docs for all the options. `open` returns an open file object.

It's worth familiarizing yourself with these functions. We use a few of them through the book, and they can be extremely useful.

This list of built-in functions includes the type objects of the datatypes we've already looked at. These create new objects of their type. For example, `int` takes numbers or strings as input and returns a new integer.

```
>>> int(3.2)
3
>>> int('3')
3
```

The other constructors do a similar job, creating new objects from input you provide.

Functions are the bread and butter of procedural and functional programming. In recent years,[13] a paradigm called object-oriented programming has arisen, building on the principles of procedural programming. Object-oriented programming allows you to combine data with functions (called methods) for working on the data. The basic element in object-oriented programming is a programming construct called the class, which we turn to now.

2.2.5 Classes

At the start of this chapter, we looked at the built-in datatypes. These are relatively simple objects, such as strings or integers, which represent a value. As well as their value, most[14] of the built-in types also have methods associated with them.

You use these methods for performing common operations; they're a bit like the built-in functions. For example, strings have methods to return a copy of themselves in lowercase or to remove whitespace from the start and end. The key thing is that the built-in types only have methods that are relevant to their type. Numbers don't have a `lower` or a `strip` method.

The selection of available methods is determined by the type of the object and its class. You can define your own classes to create new kinds of objects.[15]

NOTE Classes are the blueprints for creating new kinds of objects. Once you've defined a class, you can create as many individual objects from the blueprint as you want.

New classes are not only for representing data but also for structuring programs and providing a framework. Well-written classes will make your program easier to understand; but, more importantly, they're a thing of beauty and elegance!

Methods are attributes of objects that you call like functions—they can take arguments and return values. All object attributes are accessed using the dot syntax.

---

[13] Well, Simula from the 1960s is widely regarded as the first object-oriented language although the term was first applied to Smalltalk. By the mid 1990s, OOP had become the dominant programming methodology, largely credited to the success of C++.

[14] Not surprisingly, the `NoneType` doesn't have many useful methods on it…

[15] For a longer introduction to objects and classes, visit my (Michael's)article "Object-Oriented Programming with Python" at http://www.voidspace.org.uk/python/articles/OOP.shtml.

The following example uses the string method `startswith`. This method takes an argument and returns `True` if the string starts with the argument you supplied.

```
>>> string = 'Hello world'
>>> string.startswith('Hello')
True
```

Not all attributes are methods. Attributes can also be values, properties, or any other object. The normal way to access attributes though, whatever they may be, is with the dot syntax.

So how do you create your own classes? The answer is the `class` statement. We talk about classes being *declared*. A typical line that commences a class declaration is shown in figure 2.5.

Between the parentheses following the class name is a list of classes that your class inherits from. If your class inherits from another class, then it automatically gains the attributes and methods of that class. If you need several classes that do similar things, then you may be able to put the common parts into a single class and then have your other classes inherit from it. Classes that you inherit from are called base classes. Classes can have more than one base class—Python supports multiple inheritance—but this can be a recipe for confusion; it's usually best avoided if possible.

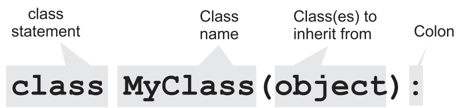


Figure 2.5 A class declaration

Old-style classes

Classes which inherit from `object` (directly or by inheriting from another class that inherits from `object`) are called new-style classes. This name comes from the revolution to the Python type system that happened in Python 2.2.

You can create classes that don't inherit from `object`;[16] these are called old-style classes, and they don't have the parentheses after the class name.

```
class OldStyleClass:
```

You can't use properties in old-style classes (something we'll come to later), and they're considered obsolete, disappearing completely in Python 3.0.

Indented inside the class statement are your methods and any class attributes. To write methods, you need to know about two things first: `self` and `__init__`.

Every method you write needs to take an extra argument. The individual objects created from classes are called *instances*. Individual strings are instances of the `str` or `unicode` class. For the code inside your methods to access the instance it belongs to, it needs a reference to the instance. This is passed in as the first argument to methods;

[16] At least don't inherit directly. Old-style classes are instances of the `ClassType` metaclass.

the Python convention is to call this argument `self`. `self` is the equivalent of `this` in C#, except that you need to explicitly declare it in the method signature.

You also need to know that constructor methods are declared in Python with the name `__init__`. `__init__` is entirely optional; you don't need to write a constructor if your class doesn't require one.

NOTE The constructor of a class is a method called when new objects are created from the class. New instances are created (or *instantiated*) by calling the class, and passing in any arguments required by the constructor.

Method names that begin and end with a double underscore (spoken as *dunder*, which is much quicker to say) often have a special purpose in Python; they're called on your objects by the interpreter. You'll encounter several of these *magic methods* as we progress.

When you create a new instance by calling the class, the constructor is called and the new instance returned.

```
>>> class MyObject(object):
...     def __init__(self, value):
...         self.value = value
...
>>> myObject = MyObject(10)
>>> print myObject.value
10
```

When the instance is created, `__init__` is called with a reference to the new instance (`self`) and the argument passed in. `__init__` then sets the `value` attribute on the instance to 10.

Classes can have attributes too; this is useful because class attributes are shared between all instances.

```
>>> class SimpleObject(object):
...     classAttribute = object()
...
>>> instance1 = SimpleObject()
>>> instance2 = SimpleObject()
>>> instance1.classAttribute is instance2.classAttribute
True
```

Note that in this example a constructor isn't defined, so you inherit the default one from `object`, which does nothing.

Python doesn't have true private or protected attributes. By *convention*, any method or attribute name that starts with a single or a double underscore isn't part of the public API. Exceptions are the magic methods, whose names start and end with double underscores. You don't often call these directly anyway, except perhaps in subclasses calling up to their parent classes. Attribute or method names that have a leading double underscore are treated specially by the interpreter. The names are mangled[17] to

[17] They're mangled by adding an underscore followed by the class name to the start of the name.

make it harder to access them from outside the class. From inside the class, the names can be used normally; from outside, they can only be accessed via the mangled name.

We've now covered the basic constructs in Python, the mortar and the bricks of the language. Functions and classes are the most essential building components of any Python program. By now, you should be starting to get an overview of Python and how it works; for example, methods are really just special cases of functions. One consequence of this is that you can replace methods on classes with alternative functions at runtime, something useful for testing. The more of an overview you have, the easier it is to understand the details of what's going on.

The goal of this chapter is to allow you to create an IronPython program that takes full advantage of the .NET platform. Many aspects of the .NET framework are similar to Python, such as the basic class structure with methods and attributes. To create Python programs, you need to understand the objects and patterns you'll encounter from a Python point of view. You also need to know enough Python to be able to create applications. Before we can dive into .NET, you need a few more raw materials; think of these as the timbers, ties, and plastics of the house that is the Python language. In the next section, we look at a few further aspects of Python necessary to our construction work.

2.3 Additional Python features

If you've used languages like VB or C# before, then most of the syntax covered so far should be at least familiar. The core language features of Python aren't large; Python is *small enough to fit your brain.* This section will look at a few more features of Python, the semantics of dividing programs into modules and importing from them, scoping rules, decorators, and so on. We look at exception handling first.

2.3.1 Exception handling

Exceptions indicate that some kind of error has happened. They can be raised by the operating system when you try something like accessing a file that doesn't exist, or by the Python runtime when you do something daft like adding strings to numbers.

You can also raise exceptions yourself, to indicate that an error condition has been reached.

If you don't handle exceptions, then your program will literally stop with a crash. Catching exceptions in Python is done with the `try:` … `except:` block.

```
>>> 'a string' + 3
Traceback (most recent call last):
TypeError: Cannot convert int(3) to String
>>> try:
...   'a string' + 3
... except:
...    pass
...
```

When used like this, the `except` clause catches all exceptions. This is bad practice; you should only catch the specific types of exceptions that you're expecting. Catching and

silencing exceptions you aren't expecting can allow bugs in your code to pass unnoticed, and cause problems that are very difficult to debug.

In Python, you can specify which types of exceptions you want to catch after the `except`. You specify either a single exception type or a tuple of exceptions. You can also chain multiple `except` clauses to provide different blocks of code to handle different types of exceptions. You can also provide a block of code to be executed only if no exceptions occur, by ending your `except` clauses with an optional `else` clause.[18]

```
>>> try:
...     do_something()
... except (OSError, IOError):
...     print 'Looks like the path was invalid'
... except TypeError:
...     print 'Why did this happen'
... else:
...     print 'It all worked out fine'
...
```

Sometimes you need to keep hold of the exception instance to check attributes or the message. The string representation of exception instances is the error message. You can bind the exception to a name within the `except` clause; this can be useful for logging.

```
>>> try:
...     'a string' + 3
... except TypeError, e:
...     print e
...
Cannot convert int(3) to String
```

In some circumstances you may wish to catch the error and then re-raise it. If you use the `raise` statement on its own, it re-raises the last exception that occurred.

For a slightly different use case, `try:` … `except:` has a companion—the `try:` … `finally:` block. You use `finally` rather than `except` when you need to guarantee that some code runs, whether or not the preceding block was successful. For example, you might want to use `finally` when closing a database connection or an open file. If an exception is raised, the `finally` block is executed, and then the exception is re-raised.

The next example illustrates the `finally` block, and also the other side of catching exceptions—raising exceptions.

```
>>> try:
...     raise Exception("Something went wrong")
... finally:
...     print 'This will be executed'
...     print 'whether an exception is raised'
...     print 'or not'
...
This will be executed
whether an exception is raised
or not
Traceback (most recent call last):
Exception: Something went wrong
```

[18] `OSError` and `IOError` are built-in Python exception types.

finally, except, and the with statement

In Python 2.4 (IronPython 1),[19] you can't have an `except` clause and a `finally` clause in the same block.

In Python 2.5 (IronPython 2), you can have both `except` clauses and a `finally` clause in the same block. If an exception is caught by an `except` clause, the code in the `finally` clause is still executed.

Another common pattern used in Python 2.5, which can often replace use of `finally`, is to use the `with` statement. This is similar to the `using` statement in C#, and allows you to use a resource with its cleanup guaranteed even if an exception is raised.

The `raise` statement raises exceptions. There are several different variations of raise, but the only one you really need to know is the form used in the example.

```
raise ExceptionType(arguments)
```

The base exception class in Python is `Exception`.

IronPython exceptions

When a .NET exception percolates through to IronPython, it's wrapped as a Python exception. You can catch a lot of .NET exceptions using the equivalent Python exceptions. For example the .NET `MissingMemberException` equates to the Python `AttributeError`.

You can see a full list of Python exceptions at http://docs.python.org/lib/module-exceptions.html.

You can find a mapping of .NET exceptions to Python exceptions at http://www.codeplex.com/IronPython/Wiki/View.aspx?title=Exception%20Model. When catching exceptions, you can use either type, and you'll get the corresponding Python or .NET exception.

That finishes it off for exceptions, at least for the moment. We've already talked at length about how Python can take advantage of programming styles like functional programming. Understanding the scoping rules is important for any programming, but it's particularly vital to functional programming.

2.3.2 Closures and scoping rules

Scoping rules are the way that a language looks up names that you use within a program. Python uses *lexical scoping*.[20] Your code can access names defined in any enclosing scopes, as well as all global values.

---

19 *Unless* you enable Python 2.5 compatibility.

20 The Python exception to full lexical scoping is that you can't rebind names in an enclosing scope.

But first, we probably need to explain what a scope is. Let's look at global and local scopes .

```
>>> x = 3                ❶
>>> def function1():
...     print x          ❷
...
>>> def function2():
...     x = 6            ❸
...     print x
...
>>> function1()
3
>>> function2()
6
```

This code segment defines the name `x`. Because it isn't inside a function or method body, `x` is defined in the global scope ❶. `function1` uses the name `x` ❷, which isn't defined inside it. When Python encounters the name and can't find it locally, it looks it up in the global scope.

Inside `function2` the name `x` is also defined locally ❸. When you call `function2` and the Python interpreter encounters the name `x`, the first place it looks is the local scope. The local name `x` shadows the global variable.

Things get interesting when you introduce another scope. If a function has a function defined inside it, the outer function is said to *enclose* the inner function. When the inner function uses a name, the name is first looked for in the scope of the inner function, its local scope. If it isn't found, then Python will check the enclosing scope. This applies to as many levels of nesting as you want; the outer scopes will be searched all the way up to the global scope. If the name isn't found anywhere, then a `NameError` exception is raised.

```
>>> x = 7
>>> y = 4
>>> def outerFunction():
...     x = 5
...     def innerFunction():        <-- Defines function inside outerFunction
...         print x, y
...     innerFunction()             <-- Calls new function
...
>>> outerFunction()
5 4
```

When you call `outerFunction`, it defines the function `innerFunction`. This function exists in the local scope of `outerFunction`. (Try calling `innerFunction` from the global scope.) `outerFunction` calls `innerFunction`; this prints the values `x` and `y`. Neither of these is local to `innerFunction`. `x` is defined inside `outerFunction`, shadowing the global `x`. `y` is only defined in the global scope.

When you call a function, the names inside it are looked up from the enclosing scope where the function was *defined*, not from where it's used. Your outer functions can return a reference to an inner function that can then be used far from where it

was created. It will still use names from the scope(s) that enclosed it when it was defined. To illustrate this, let's use a classic example and create an `adder` function.

```
>>> def makeAdder(value):
...     def newAdder(newValue):         ❶
...         return value + newValue      ❷
...     return newAdder
...
>>> adder = makeAdder(7)         ❸
>>> adder(3)
10
>>> adder(-4)
3
```

You pass in a value to the `makeAdder` function, and it returns a `newAdder` function ❶ that adds numbers to the original value that you passed in ❷. When you call `makeAdder` with an argument ❸, the argument is bound to the name `value`, which is local to the scope of `makeAdder`. You can reuse `makeAdder` to create as many adder functions as you like. Every time you call it a new scope is created.

Several parts of Python show the influence of functional programming. One favorite of mine is the list comprehension, which is a particularly elegant construct. List comprehensions also get used a lot in Python, so it would be impossible to have a tutorial that doesn't mention them.

2.3.3 *List comprehensions*

List comprehensions allow you to express several lines of code in a single readable line. This is a concept borrowed from the language Haskell, that allows you to combine a `for` loop and an optional filter in a single construct. If you're brave, you can combine several loops into a single list comprehension, but this tends to be a bit taxing on the brain.

List comprehensions are so named because they create lists. Consider the following simple segment of code:

```
>>> result = []
>>> input = [1, 2, 3, 4, 5]
>>> for value in input:
...     result.append(value * 3)
...
>>> result
[3, 6, 9, 12, 15]
```

This constructs a list (called `result`) from all the members of `input`, multiplying each value by three.

This segment of code can be written in a much more concise way using a list comprehension.

```
>>> input = [1, 2, 3, 4, 5]
>>> result = [value * 3 for value in input]
>>> result
[3, 6, 9, 12, 15]
```

The expression `value * 3` is calculated for each member of `input`, and the `result` list is built. The two forms of this code, using a loop and using a list comprehension, are functionally identical.

List comprehensions also allow you to include filters. Suppose you only wanted to use the values from the input that are greater than two; you can put this condition into the body of the list comprehension.

```
>>> input = [1, 2, 3, 4, 5]
>>> result = [value * 3 for value in input if value > 2]
>>> result
[9, 12, 15]
```

A list comprehension is itself an expression, and so you can use one anywhere an expression can be used.

We haven't covered every possible variant of Python syntax, but you should have learned enough to give you a good grounding to be able to start exploring. The final sections in this chapter are about organizing programs into libraries called *modules* and *packages*. The mechanisms for working with modules and packages are useful for organizing your own programs, but they're also the same mechanisms by which you access the .NET libraries.

2.3.4 Modules, packages, and importing

The last thing you want when programming is to have all your code contained in a single monolithic file. This makes it almost impossible to find anything. Ideally, you want to break your program down into small files containing only closely related classes or functionality. In Python, these are called *modules*.

NOTE A module is a Python source file (a text file) whose name ends with .py. Objects (names) defined in a module can be imported and used elsewhere. They're very different from .NET modules, which are partitions of assemblies.

The import statement has several different forms.

```
import module[21]
from module import name1, name2
from module import name as anotherName
from module import *
```

Importing a module executes the code it contains and creates a module object. The names you've specified are then available from where you imported them.

If you use the first form, you receive a reference to the module object. Needless to say, these are first-class objects that you can pass around and access attributes on (including setting and deleting attributes). If a module defines a class `SomeClass`, then you can access it using `module.SomeClass`.

---

[21] This can also be written using the third form `import module as SomethingElse`.

If you need access to only a few objects from the module, you can use the second form. It imports only the names you've specified from the module.

If a name you wish to import would clash with a name in your current namespace, you can use the third form. This imports the object you specify, but binds it to an alternative name.

The fourth form is the closest to the C# `using` directive. It imports all the names (except ones that start with an underscore) from the module into your namespace. In Python, this is generally frowned on. You may import names that clash with other names you're using without realizing it; when reading your code, it's not possible to see where names are defined.

Python allows you to group related modules together as a *package*. The structure of a Python package, with subpackages, is shown in figure 2.6.

NOTE A package is a directory containing Python files and a file called __init__.py. A package can contain subpackages (directories), which also have an __init__.py. Directories and subdirectories must have names that are valid Python identifiers.

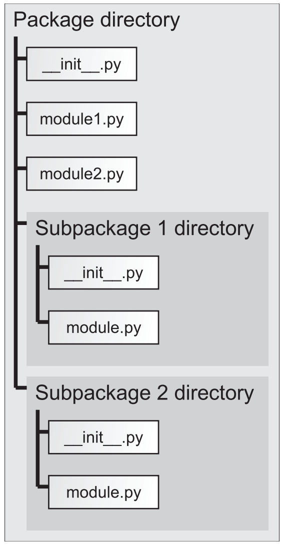


Figure 2.6 The structure of a Python package on the filesystem

A package is a directory on the Python search path. Importing anything from the package will execute __init__.py and insert the resulting module into `sys.modules` under the package name. You can use __init__.py to customize what importing the package does, but it's also common to leave it as an empty file and expose the package functionality via the modules in the package.

You import a module from a package using dot syntax.

```
import package.module
from package import module
```

Packages themselves may contain packages; these are subpackages. To access subpackages, you just need to use a few more dots.

```
import package.subpackage.module
from package.subpackage import module
```

Python also contains several built-in modules. You still need to import these to have access to them, but no code is executed when you do the import. We mention these because one of them is very important to understanding imports. This is the `sys` module.[22]

When you import a module, the first thing that Python does is look inside `sys.modules` to see if the module has already been imported. `sys.modules` is a

[22] You can find a reference to the `sys` module at http://docs.python.org/lib/module-sys.html. This module exposes some CPython implementation details, such as access to Python stack frames, which don't exist in IronPython. Most of the `sys` module is available, though.

dictionary, keyed by module name, containing the module objects. If the module is already in `sys.modules`, then it will be fetched from there rather than re-executed. Importing a module (or name) from different places will always give you a reference to the same object.

If the module hasn't been imported yet, Python searches its path to look for a file named module.py.[23] If it finds a Python file corresponding to the import, Python executes the file and creates the module object. If the module isn't found, then an `ImportError` is raised.

The list of paths that Python searches is stored in `sys.path`. This is a list of strings that always includes the directory of the main script that's running. You can add (or remove) paths from this list if you want.

Some Python files can be used both as libraries, to be imported from, and as scripts that provide functionality when they're executed directly. For example, consider a library that provides routines for converting files from one format to another. Programs may wish to import these functions and classes for use within an application, but the library itself might be capable of acting as a command-line utility for converting files.

In this case, the code needs to know whether it's running as the main script or has been imported from somewhere else. You can do this by checking the value of the variable `__name__`. This is normally set to the current module name *unless* the script is running as the main script, in which case its name will be `__main__`.[24]

```
def main():
   # code to execute functionality
   # when run as a script

if __name__ == '__main__':
   main()
```

This segment of code will only call the function `main` if run as the main script and not if imported.

When you re-import a module, you get a new reference to the already-created module object. This can be a pain when you're actively developing a module and want to play with it from the interactive interpreter. If you want to force the interpreter to use the latest version, you can use the `reload` function and then re-import it.

```
>>> import MyModule
>>> reload(MyModule)            <- Call when module changes
<module 'MyModule' from 'MyModule.py'>
>>> import MyModule             <- Re-imports module
```

Calling `reload` *doesn't* rebind names that point to objects from the old module. You have to do this yourself after reloading—hence, the need to import again.

---

[23] As well as searching for a corresponding Python file, IronPython looks for a package directory, a built-in module, or .NET classes. You can even add import hooks to further customize the way imports work.

[24] You can access the namespace of the main script by doing `import __main__`.

Another useful feature of modules shared with several types of objects is the docstring.

2.3.5 Docstrings

If you cast your mind back to the end of the first chapter, you'll recall the useful built-in function `help`. This is for working with the interactive interpreter; it prints some useful information about an object to the console. For Python modules, functions, classes, and methods, it will include any docstring that you've defined. So what is this esoteric creature, the docstring?

A docstring is a string written inline with an object and that isn't assigned a name. This is then available to the `help` function, and lives attached to the object as the `__doc__` attribute. The following segment illustrates this a bit more clearly, using a function as an example:

```
>>> def adder(val1, val2):
...    "This function takes two values and adds them together."
...    return val1 + val2
...
>>> help(adder)
Help on function adder in module __main__

 | adder(val1, val2)
 |     This function takes two values and adds them together.
>>> adder.__doc__
'This function takes two values and adds them together.'
>>>
```

If you want docstrings that contain multiple lines, you can use triple-quoted strings.

Docstrings aren't *just* useful for interactive help; they're guides for anyone reading your source code and are also used by automatic documentation tools[25] for creating API documentation from source code.

At last, you know enough to create large and well-structured IronPython programs, as long as you don't need to access much .NET functionality. Before you rush off to try this, we'd like to save you some time. CPython comes with a large standard library of packages and modules for common programming tasks. Before you start coding, it's well worth checking to see if at least part of your task is already written for you. The next section talks about the standard library, introduces you to a few of the more commonly used modules, and explains how to use the library with IronPython.

2.3.6 The Python standard library

Python comes with *batteries included*. This statement refers to the standard library that accompanies CPython. Many of the modules it provides are pure-Python modules, but some of them are extension modules written in C.

---

[25] Like Epydoc: http://epydoc.sourceforge.net.

NOTE An open source project called Ironclad,[26] created by Resolver Systems, reimplements the Python C-API in C# with the goal of allowing you to import and use Python C extensions from IronPython.

Unfortunately, extension modules written in C won't work directly in IronPython. Other modules, even in the standard library, rely on implementation details or undocumented features of Python. Some of these incompatibilities between Python and IronPython have already been fixed, but some are very difficult cases.

That's the bad news. The good news is that a large proportion of the standard library works fine. The IronPython team, along with the Python community, has put a lot of effort into ensuring that as much of the standard library as possible works with IronPython; in some cases, alternatives to C extension modules have been created. Existing code that uses the standard library stands a better chance of working, and these modules are available to use within your code.

NOTE The standard library is subject to the liberal Python License.[27] You are free to distribute it (or parts of it) along with your application as long as you include the copyright notice.

The first step in using the standard library is obtaining it. The IronPython 2 msi installer comes with the Python 2.5 standard library, and makes it automatically available for import.[28]

An alternative way of getting hold of the standard library is downloading and installing CPython. Go to www.python.org/download/ and download the latest version of Python 2.5.[29]

On Windows, the default install location is C:\Python25. The standard library itself will be located in C:\Python25\Lib.

You then need to expose the library to IronPython. There are three convenient ways of doing this. The first is best for use on a single machine, the second for distributing applications, and the third for quick one-off experimentation.

- Set an environment variable `IRONPYTHONPATH` to the location of the standard library directory.
- Copy the whole standard library into a directory accessible by your application. You'll need to add this directory to `sys.path` at runtime.
- Start IronPython while the current directory is C:\Python25\Lib.

You can manually add the standard directory library at the interactive interpreter with the following steps:

```
>>> import sys
>>> sys.path.append(r'C:\Python25\Lib')
```

[26] See http://code.google.com/p/ironclad.
[27] See http://www.python.org/download/releases/2.5.2/license/.
[28] IronPython automatically adds the directory `sys.exec_prefix + 'Lib'` to `sys.path`.
[29] Version 2.5.2 at the time of writing.

Notice that you use a raw string to add the path; otherwise, the backslashes will be interpreted as escape characters.

Table 2.3 Useful standard library modules

Module	Purpose
`sys`	Provides system-specific settings like the path, executable name, and command-line arguments.
`os`	Provides functions for interacting with the operating system, including launching processes.
`os.path`	Provides functions for working with files, directories, and paths.
`re`	Regular expression library.
`math`	Floating point math routines.
`random`	Generates random numbers and for making random selections.
`time`	For working with times and formatting date strings.
`unittest`	An extensible unit test framework.
`optparse`	A library for parsing command-line arguments.
`cPickle`	Provides object persistence. Serialize Python or .NET objects as text or binary.
`decimal`	Provides support for decimal floating point arithmetic.
`cStringIO`	Implements a file-like class (`StringIO`) that reads and writes a string buffer.
`collections`	High-performance container datatype, the deque.
`itertools`	A number of iterator building blocks.
`types`	Provides access to type objects for components such as classes, functions, and methods.

There are many modules and packages in the Python standard library. We've listed some of the commonly used ones in table 2.3. Some of these modules, such as `re` and `cStringIO`, are implemented in C for CPython. Many of them[30] have been implemented by the IronPython team as built-in modules (written in C#) so that these modules are available to IronPython. You can find a list of all of them at http://docs.python.org/modindex.html.

Python isn't a big language, nor is it difficult to learn, but we've still covered a lot of material in this chapter. The best way of getting it ingrained, of course, is to use it. In the next chapter, you'll be putting what you've learned here to practical use; you'll also learn about the .NET framework, which comes with its own standard library.

[30] All the modules listed in table 2.3 work with IronPython, but not every C extension in the standard library has been ported.

2.4 Summary

There's a great deal to Python that we haven't looked at; but, if we've been doing our job, then what you've learned will give you a good foundation. Python emphasizes a readable style of programming with the use of English keywords wherever possible. You now know the bare bones of the Python programming language—the frame of the house without the furnishings that make a house a home. We completed the chapter by looking at how to use the Python standard library with IronPython. The standard library provides a large set of Python modules for you to use with your code to supplement what's available through .NET.

Any program you write will use classes that you create yourself, along with ones from the Python standard library and the .NET framework. Although we've covered the syntax needed to write classes, writing ones that make the best use of Python takes practice—you need to write some.

More importantly, we haven't covered the use of the classes from the .NET framework; this, after all, is the reason to be using IronPython. In the next chapter, you'll be able to use the knowledge gained in this chapter, and put it to work with the power of .NET. In the process, you'll be getting some practice at writing your own Python classes.

3 .NET objects and IronPython

This chapter covers

- Reading the MSDN documentation
- .NET types: Classes, structures, enumerations, and collections
- Delegates and event handlers in IronPython
- Subclassing .NET classes

.NET is a large platform and has a great deal to offer programmers. The framework is extremely broad; the classes it provides cover almost every aspect of programming that you can imagine. The IronPython .NET integration makes it easy to use these classes directly from IronPython code; you'll rarely do type conversion between Python types and .NET ones. Not only are the .NET objects straightforward to use and configure from within IronPython, but they're also integrated into the Python infrastructure so that you can use Python idioms to work with them.

To understand how to use the .NET types available in the framework, you'll need to be able to read the .NET documentation. In this chapter, you'll create some basic applications using Windows Forms and translate the Microsoft documentation into

IronPython along the way. Windows Forms is a mature library for building user interfaces for desktop applications. We could have picked almost any of the different libraries provided by .NET, but Windows Forms is a great library to work with, and it also makes for some very visual examples!

3.1 Introducing .NET

The .NET framework consists of 112 assemblies and 935 namespaces.[1] .NET 3.0, which introduces several important new libraries including the Windows Presentation Foundation, builds on .NET 2.0 rather than replacing it. Namespaces are the libraries that provide classes and other .NET types such as enumerations and structures. With these, you create applications,[2] work with XML and databases, communicate with the computer system, or access external applications such as Word and Excel.

The most fundamental libraries of the framework are known as the Base Class Libraries (BCL).[3] The BCL includes namespaces like `System`, `System.Collections`, and `System.IO`. Higher level libraries, such as Windows Forms, are built on top of the base class libraries. Figure 3.1 shows this structure—from the operating system on which the CLR runs, through the base class library, and up to the intermediate and higher level libraries.

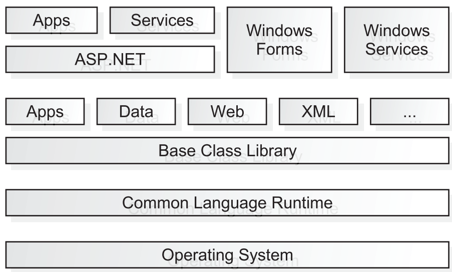


Figure 3.1 An outline of the structure of the .NET framework

We're going to start exploring the .NET framework with a handful of classes from the Windows Forms namespace.

3.1.1 Translating MSDN documentation into IronPython

We've done enough talking about using .NET classes; it's time to put what you've learned into action. Let's start by creating a simple Windows Forms application, using classes from the `System.Windows.Forms` assembly.

Windows Forms is a rich and mature GUI framework. It contains an *extraordinary* number of different components for building applications, including sophisticated data-bound controls.[4] You can see a small selection of the components available in figure 3.2.

Windows Forms controls include standard components such as labels, text boxes, radio buttons, and check boxes. It also has many more complex controls such as rich text boxes (as shown in the figure with the C.S. Lewis text), calendars, and data tables.

---

1 As of .NET 3.5 SP1. Source: http://blogs.msdn.com/brada/archive/2008/08/18/what-changed-in-net-framework-3-5-sp1.aspx.

2 Command-line applications, Windows applications, and web applications or services.

3 See http://msdn2.microsoft.com/en-us/netframework/aa569603.aspx.

4 *Control* is the .NET word for a GUI element. In other GUI libraries, they're often referred to as *widgets*.

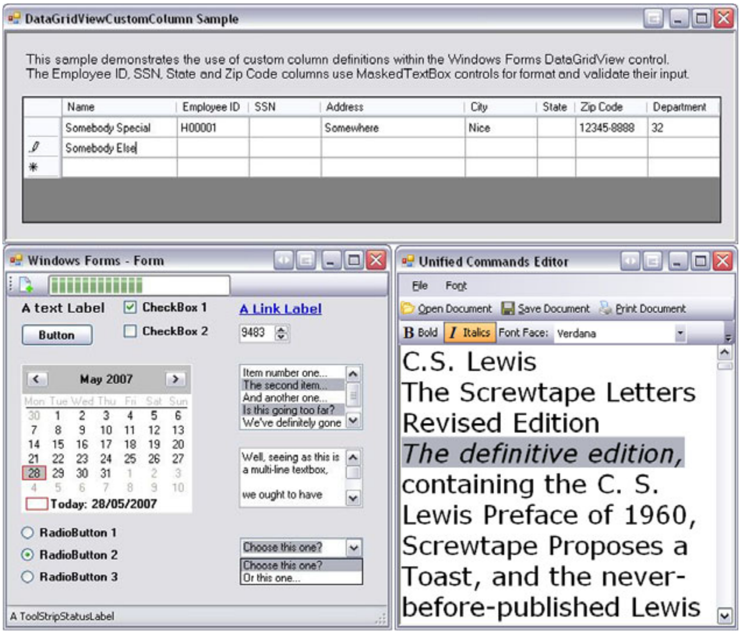


Figure 3.2 Example applications showing off a few Windows Forms controls

Recently, a new user-interface library has been added to the .NET framework: WPF. This has a more sophisticated appearance[5] and some interesting integration with XAML and vector graphics, but has fewer controls and is less mature than Windows Forms. Windows Forms is still the standard for building business applications.

To build a Windows Forms application, the first two components we need are the `Application` and `Form` classes. To configure and use these classes, you should first look up their attributes in the documentation.

The documentation for the framework libraries is in the Microsoft Developer Network (MSDN) library. It contains references for C#, the .NET framework, and other Microsoft programming resources.

The documentation for the `Application` class is located at the following URL:

http://msdn2.microsoft.com/en-us/library/system.windows.forms.application.aspx

Here you'll find general information about the class and some code examples. More useful is the Application Members page, which lists all the attributes of the `Application` class. Every framework class has a page like this, listing all its members with a brief description.

The top of the Application Members page tells you that the `Application` class provides *static* methods. Static methods are called on the class rather than on an instance. Here you'll find the method `Run`, which is the one we need. If you click this link, it will take you to the documentation for the static method `Run`.

[5] It's also easier to skin controls to customize their appearances.

This page shows that `Application.Run` can be called in three different ways. .NET allows classes to have multiple versions of methods, including constructors. Multiple methods and constructors are called overloads; the type of objects you pass in determines which overload is called. The three different call signatures of `Run` are as follow:

- `Application.Run()`
- `Application.Run(ApplicationContext)`
- `Application.Run(Form)`

Because we don't want to mess around with application contexts right now, and we *do* want to display a form, we use the third overload. This overload requires an instance of the `Form` class.

3.1.2 The Form class

The `Form` documentation is located at the following URL:

http://msdn2.microsoft.com/en-us/library/system.windows.forms.form.aspx

If you browse over to the Form Members page, you'll find the link the same way you found the link to Application Members. You'll be configuring the `Form` instance through the public properties, which are all listed on this page; you've already used a couple of them in the interactive interpreter example. At the top of the page, you can see a link to the Form Constructor.

```
public Form()
```

This bit of code tells you that the constructor is public (which is great news because you want to use it) and takes no arguments.

Now you know how to start the application loop and create a form, which is the most trivial Windows Forms application possible. Let's put this into practice.

To make it a bit more interesting, first set a title on the form. Turn back to the Form Members page of the MSDN document and look at the properties. There you'll see a property named `Text`. This is yet another link; clicking it takes you to a page with information about the `Text` property.

The code examples there tell you what you need to know about this property.

From the C #and the Visual Basic examples (shown in figure 3.3), you can tell that this property takes a string and that it can be both set *and* retrieved. Some properties are read-only, but not this one (and there are a few that are *write*-only—which is pretty weird and also pretty rare).

Visual Basic (Usage)

```
Dim instance As Form
Dim value As String

value = instance.Text

instance.Text = value
```

C#

```
public override string Text { get; set; }
```

Figure 3.3 The C# / VB.NET examples for the `Form.Text` property

NOTE When we refer to the current versions of IronPython, we mean the versions at the time of writing: IronPython 1.1.2 and 2.0. All the IronPython code in this book should work with both versions of IronPython; the differences between IronPython 1 and 2 will be shown in the examples. If there are limitations, we've *tried* to refer to specific versions.

Execute the following lines of IronPython code at the interactive interpreter:

```
>>> import clr
>>> clr.AddReference('System.Windows.Forms')
>>> from System.Windows.Forms import Application, Form
>>> form = Form()
>>> form.Text = 'Hello World'
>>> Application.Run(form)
```

If you've entered it correctly, you should see a plain form with the title set to Hello World, looking similar to figure 3.4.

An important thing to note from this example is that, when you set the `Text` on the form, you used a string literal from IronPython. Strings created in IronPython are Python strings; they have all the usual Python string methods, but they're also .NET strings.



Figure 3.4 A `Form` with the title (`Text` property) set

> ### Another way to set properties
> You can also set properties by using the constructor of the control. The alternate form of the previous example looks like the following:
>
> ```
> form = Form(Text='Hello World')
> ```
>
> This uses keyword arguments and looks a bit more Pythonic.[6]

IronPython strings, numbers, and Booleans aren't just similar to their .NET equivalents; they *are* .NET types.

NOTE You've already met the Python built-in types. Strings, numbers, and Booleans are examples of the .NET built-ins.[7]

This example may seem like a trivial one, but that's only because it is! You haven't done anything that you didn't do in the interactive interpreter example at the end of chapter one. The important thing to take away is how to navigate the MSDN documentation. In this book, we can cover only a tiny proportion of the available classes, but the technique for finding out the information you need is identical.

---

[6] *Pythonic* is a subjective word used to describe whether or not a piece of code holds with recommended Python idioms.

[7] See http://msdn2.microsoft.com/en-us/library/ya5y69ds.aspx for a reference to all the C# built-in types.

After running the example, you'll see the form appear. Starting the application loop shows the form you pass in. A form on its own is a bit dull, though; let's add some more controls to the form. Along the way, you'll meet some more of the .NET types.

3.2 Structures, enumerations, and collections: .NET types

The .NET framework is a different programming environment than Python, and it uses types that aren't native to the Python world. These include structures, enumerations, and different container types. Because of the deep level of integration between IronPython and .NET, you can use these types directly in IronPython. In this section, you'll use several of these types in the context of working with Windows Forms controls.

3.2.1 Methods and properties inherited from Control

Adding new GUI elements to a form is done in the same way regardless of the element you want to add. A wide range of these elements are available, each of which is represented by a different class in the `System.Windows.Forms` namespace.

It's possible to create forms and then add and configure components, using the Visual Studio designer.[8] You can see in figure 3.5 that Visual Studio will generate

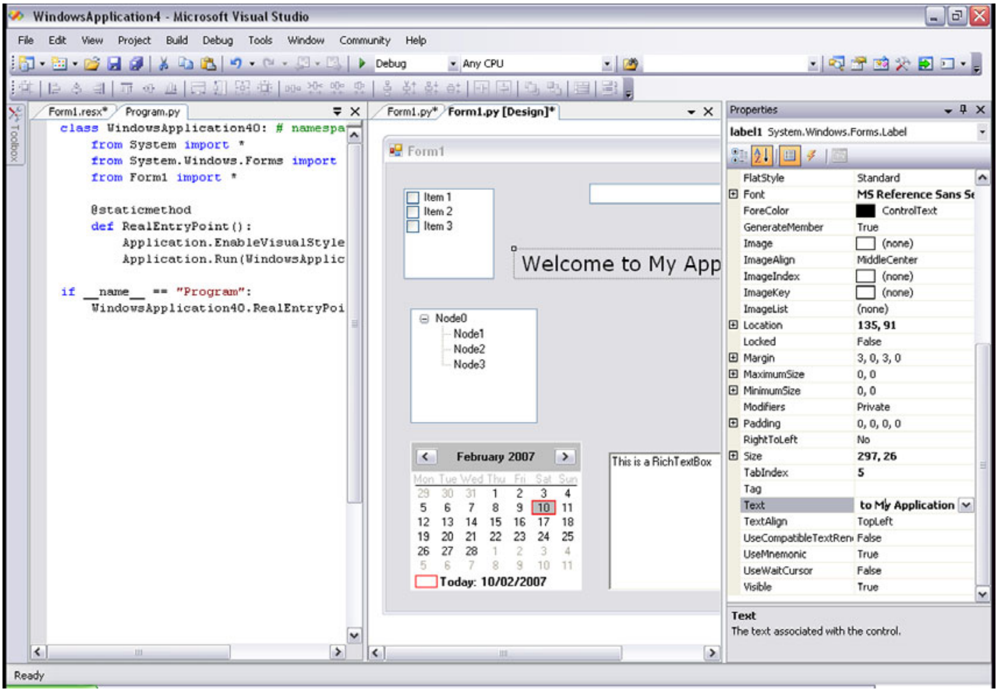


Figure 3.5 Visual Studio in designer mode, with an IronPython project. IronPython code is on the left, and control properties are on the right.

[8] Generating IronPython from Visual Studio 2005 requires the full version of Visual Studio and the SDK; it can't be done with Visual Studio Express. Instead, you can use IronPython Studio, which is based on Visual Studio 2008 and works as a standalone application *or* integrated into Visual Studio.

IronPython code for you, and let you configure the control properties. Whether you're using the designer or building your user interfaces by hand, you still need to understand the components to configure them.

Let's start with a nice simple component: the `Label` class. It's used for adding text labels to forms; it doesn't do a great deal, but you can configure it in a lot of ways. You can set the position and size of the label; you can also configure the color of text on the label and the background color. This configuration is done using properties common to other classes within the `System.Windows.Forms` namespace. The Windows Forms' term for GUI components, such as forms, buttons, and labels, is *controls*. A label is a control, and so are buttons and text boxes.

These classes all inherit from the `Control` class. Different categories of controls inherit from various other intermediate classes such as `ScrollableControl` or `ContainerControl`, but all the GUI elements inherit directly or indirectly from `Control`.

If you look at the Label Members page[9] on MSDN, you'll see that many of the members have *(Inherited from Control)* by them. Why is this interesting? A lot of the properties and events you'll use in the coming sections are like this. Once you've learned how to use them on a specific control, you've learned to use them on other controls as well. Some controls may override the inherited member, but the usage will still be similar or related to the standard behavior.

Properties that come into this category include the following:

- `Text`
- `BackColor`
- `ForeColor`
- `Font`
- `Size`
- `Location`
- `Height`
- `Width`

But just as a bare form is pretty pointless, so is a lonely label. Individual controls only have a point when part of a GUI. You need to know how to add controls to a parent control, and that means working with a collection.

3.2.2 Adding a Label to the Form: ControlCollection

One of the properties inherited from `Control` not mentioned in the last section is `Controls`. `Control.Controls` is a special type of collection, specifically a `ControlCollection`. Collections are .NET container classes, similar to Python lists and dictionaries.

The `ControlCollection` keeps track of the child controls on a parent and keeps them in order. When the parent is displayed, all the child controls are also displayed. You can add, remove, or reorder the controls in the collection by calling methods on

9 http://msdn2.microsoft.com/en-us/library/system.windows.forms.label_members.aspx

it. You may remember the simplest of these from the interactive interpreter example at the end of chapter one. To add a control to a `Form`, or any other parent control, use the `Add` method of the `Controls` property.

```
>>> form = Form()
>>> label = Label()
>>> form.Controls.Add(label)
```

You can see a list of *all* the methods available on the `Controls` property at the ControlCollection members page.[10] The most useful methods include the following:

- *`Clear`*—Removes all controls from the collection
- *`Remove`*—Removes the specified control from the control collection
- *`RemoveAt`*—Removes a control from the control collection at the specified index

You can see from the `RemoveAt` method that the control collection keeps controls by index. It uses the standard .NET indexing mechanism by implementing the `IList` interface. In Python terms, this means that you can index it like a list (using positive indexes only).

```
>>> label is form.Controls[0]
True
```

You can also iterate over the controls in the collection—not because `ControlCollection` implements `IList`, but because it implements the `IEnumerable` .NET interface. IronPython ensures that enumberable objects from other .NET languages are iterable in IronPython.

`ControlCollection` has two more useful attributes, the property `Count` and the method `Contains`. You don't need to use these directly in order to take advantage of them.

.NET objects that contain other objects often have a `Length` or a `Count` property, which tells you how many members they contain. In Python, you get the length of a container using the built-in function `len`.

```
>>> len(form.Controls)
1
```

The `Contains` method checks whether a collection contains a specified member. In Python, you check for membership in a container using the `in` operator, and this is available on .NET collections.

```
>>> label in form.Controls
True
>>> newLabel = Label()
>>> newLabel in form.Controls
False
```

Before leaving this subject, we'd like to look at one more `ControlCollection` method, one you *probably* won't use very often from IronPython. `AddRange` allows you

---

[10] http://msdn2.microsoft.com/en-us/library/system.windows.forms.control.controlcollection_members.aspx

to extend the collection by adding an array of controls. Normally, you'll add controls to the collection individually, but we want to use this method to demonstrate that Python objects aren't always the direct equivalent of similar .NET objects.

.NET arrays are similar to Python lists—except that, because the languages they're used from are statically typed, arrays carry information about the type of objects they contain. An array can only contain objects of a single type. Calling a .NET method, which expects an array, and giving it a list will fail.

```
>>> form.Controls.AddRange([label])
Traceback (most recent call last):
TypeError: expected Array, got list
```

Although Python lists are similar to .NET arrays, they aren't directly equivalent; you'll need to convert your list into an array if you want to use the `AddRange` method.

We've been looking at the `ControlCollection` available on Windows Forms controls as the `Controls` property. The integration of .NET with IronPython doesn't extend only to instantiating and configuring classes. Whenever you come across.NET collections, there are various ways in which you can treat them in the same way as Python containers, but there are some differences.

We started this section by looking at how you add controls to a form. Adding to the `Controls` collection is one way, but another way has exactly the same effect. You can assign the parent control to the `Parent` property of the child control. Under the hood, the child control will be added to the parent's collection.

```
>>> form = Form()
>>> label = Label()
>>> label.Parent = form
>>> len(form.Controls)
1
```

We've thoroughly covered *how* to add a new label to a form, so let's look at some more ways to configure a label. In the process, you'll meet a new .NET type: the structure.

3.2.3 Configuring the Label: the Color structure

Our form, with a single label, is still going to look pretty dull. In this section, you'll add some color, literally.

All controls have two properties particularly related to color, `ForeColor` and `BackColor`. `BackColor` sets the background color of the control. The `ForeColor` affects elements of the control, such as the text of a label.

You specify the color using the `Color` structure from the `System.Drawing` namespace. `System.Drawing` lives in its own assembly; to use it, you need to add a reference to its assembly.

```
>>> clr.AddReference('System.Drawing')
>>> from System.Drawing import Color
```

NOTE The struct type is suitable for representing lightweight objects. Although it's always possible to represent these using classes, structs use less memory.

Structures carry small but related amounts of information when using a class isn't justified. Structures typically have named fields, but they can also have constructors and members, and even implement interfaces. In Python, you'd probably use a class or create a data structure using the built-in types; when you use structures from IronPython, it's likely to be making use of ones from .NET.

The `Color` structure has a multitude of fields, representing the .NET selection of predefined colors with weird and wonderful names.[11] To use a specific color, specify it by name.

```
>>> label.BackColor = Color.AliceBlue
>>> label.ForeColor = Color.Crimson
```

Individual colors are defined by four eight-bit values—the Red, Green, Blue, and Alpha channels—for each specified color. You can access these values on the named members of the `Color` structure.

```
>>> Color.Red.R, Color.Red.G, Color.Red.B
(255, 0, 0)
>>> Color.Red.A
255
```

If you can't find the color you need from the fantastic selection of named colors that the .NET framework offers, you can create new ones with the `FromArgb` method, which has several overloads. The most common way of using it is to pass in three integers representing the red, green, and blue components.

```
>>> newColor = Color.FromArgb(27, 94, 127)
```

In the exploration of control properties, you've now encountered several .NET types. One you haven't yet seen is the enumeration. In the next section, you'll use the `FormBorderStyle` property, which takes an enumeration.

3.2.4 The FormBorderStyle enumeration

So far, you've been configuring the humble label. The label has another property inherited from `Control`: the `BorderStyle`. It isn't *normal* to configure a border on a label. It makes more sense to configure this on a `Form`, using the `FormBorderStyle` property instead.

Unsurprisingly, `BorderStyle` and `FormBorderStyle` configure the style of border that Windows Forms draws controls with. You might want to change the border style on a panel (containing a group of controls) to make it stand out within a user interface, or you might want to make a form look more like a dialog box.

You have three different border styles to choose from; you specify the one you want using a .NET type called an enumeration.

The `FormBorderStyle` enumeration belongs in the `System.Windows.Forms` namespace. It has the following three members:

[11] As usual, you can see them all on the Color Members page.

> **Enumeration**
>
> An enumeration provides named members.[12] The underlying values of the members are constants, but the use of enumerations makes code more readable. Usually, but not always, the underlying values are 32-bit signed integers.
>
> If it makes sense for more than one member of an enumeration to be used simultaneously, then members of the enumeration can be combined in a bitwise or operation. These are called *flag enumerations*.

- *`Fixed3D`*—A three-dimensional border
- *`FixedSingle`*—A single-line border
- *`None`*[13]—No border

To change the border style of a form, you set the `FormBorderStyle` property to a member of the `BorderStyle` enumeration.

```
>>> form = Form()
>>> form.FormBorderStyle = FormBorderStyle.Fixed3D
```

This pattern is used quite a bit in Windows Forms. You'll often configure controls by setting a property, using an enumeration with the same (or similar) name as the property. Now it's time to put together everything you've learned so far and see the results.

3.2.5 Hello World with Form and Label

We've now covered quite a lot of theory. Let's put it into practice and display a form with a configured label. You'll use a few .NET types that you haven't yet seen, but it should be obvious what they're doing. Listing 3.1 creates a form that shows a label. The font, color, and position of the label are configured using the types we've been discussing.

Listing 3.1 Showing a `Form` with a `Label`

```
import clr
clr.AddReference('System.Windows.Forms')
clr.AddReference('System.Drawing')
from System.Windows.Forms import (
   Application, Form,
   FormBorderStyle, Label
)
from System.Drawing import (
Color, Font, FontStyle, Point
```

[12] Not everything that looks like an enumeration is one. The use of predefined colors, such as `Color.Red`, makes `Color` look like an enumeration—it's a struct.

[13] Normally an object member named `None` would be invalid syntax in Python. This is a limitation of the CPython parser and would be inconvenient when working with .NET, which has several enumerations with members named `None`.

```
)

form = Form()
form.Text = "Hello World"
form.FormBorderStyle = FormBorderStyle.Fixed3D
form.Height = 150

newFont = Font("Verdana", 16,           <— Creates font object
   FontStyle.Bold | FontStyle.Italic)

label = Label()
label.AutoSize = True
label.Text = "My Hello World Label"
label.Location = Point(10, 50)
label.BackColor = Color.Aquamarine
label.ForeColor = Color.DarkMagenta
label.Font = newFont

form.Controls.Add(label)      <— Adds label to form

Application.Run(form)      <— Launches application
```

In figure 3.6, you can see the result, showing a form with a configured label. The only really new part of the code in this listing is setting the font on the label.

Fonts are created using the `Font` class from the `System.Drawing` namespace. You can create fonts in several ways; `Font` has *fourteen* different ways of calling its constructor! This example uses a string to specify the font family, an integer for the size, and the `FontStyle` enumeration for the style. To make the font bold *and* italic, the style is specified with a bitwise OR of the two different `FontStyle` members.



Figure 3.6 The result of showing a `Form` with configured `Label`

```
FontStyle.Bold | FontStyle.Italic
```

This form is still pretty dull, though; it doesn't do a great deal. Once `Application.Run` has been called the event loop is in charge of the application. To make things happen, the application has to be able to respond to events.

3.3 Handling events

In the form you've created, you haven't set up any event handlers. When programming with a GUI toolkit, the usual model is *event-based programming*; this is the case for Windows Forms.

Instead of following a linear path through a program, you put in place event handlers to respond to certain events. The message loop (sometimes called the application loop or the message pump) manages these events and calls your event handlers when an event they're listening for occurs.

These events can be a great deal of things—key presses, mouse clicks, or a control receiving focus, for example. You usually have the chance to respond to events at several

different levels. You could put an event handler to respond to any mouse clicks (and possibly cancel the event); or, if the click happens over a button, you could listen to the click event on the button instead.

Events are also used throughout the .NET framework for working with networks, timers, threads, and more. Handling events from IronPython is very easy; let's explore them using the `MouseMove` event.

3.3.1 Delegates and the MouseMove event

In the pages of control members that we've looked at so far, you might have noticed the parts of the pages headed *Public Events*, *Protected Events*, and *Private Events*. These are all the different events that these controls can raise. To have your code respond to these events, you create event handlers and add them to the event.

If you (again) browse over to the Form Members page and look in the Public Events section, you'll see an event called `MouseMove`. This event allows a control to respond to mouse moves while the mouse is over the control. You could use this event to draw a rectangle on the form as the user moves the mouse—say, to highlight an area of an image.

The page you reach by following the MouseMove link from the Form Members page tells you about the event. Most straightforwardly, it tells you that this event *Occurs when the mouse pointer is moved over the control.*

More importantly, it also shows you how to use the event, but you have to dig into the C# example. In the long code example, which illustrates using various mouse events, is the following line:

```
this.panel1.MouseMove += new
System.Windows.Forms.MouseEventHandler(this.panel1_MouseMove);
```

`MouseMove` is an event on a panel. The event handler is added to the event using `+=` (add-in-place). Event handlers in C# must be delegates; for mouse events, the delegate needs to be a `MouseEventHandler`, which is defined in `System.Windows.Forms`. In C#, delegates are classes that wrap methods so that they can be called like functions. Here the delegate wraps the `panel1_MouseMove` method, which is to be called when the `MouseMove` event occurs. Luckily, it's easier to use events from IronPython than it is from C#.

You need to know how to write your event handler, and the only important detail is what arguments the handler will receive. In the C# example, the `panel1_MouseMove` method is declared as

```
private void panel1_MouseMove(object sender,
      System.Windows.Forms.MouseEventArgs e)
{ ... }
```

The event handler receives two arguments: `sender`, which is declared as being an `object` (very informative); and `event`, which is an instance of `MouseEventArgs`. This pattern is the same for all Windows Forms event handlers; they receive the sender and an event. The sender is the control that raises the event.

Not all events have particularly useful information on them; sometimes it's enough to know that the event occurred, and you can ignore the event passed into the handler. For the MouseMove event, the event can tell you the location of the mouse.

Now you need to know how to create the required event handler in IronPython.

3.3.2 Event handlers in IronPython

Python has a native function type, so the IronPython engine does some behind-the-scenes magic for you, allowing you to use functions as event handler delegates.

Creating and adding simple event handler looks like the following:

```
form = Form()
def onMouseMove(sender, event):
   print event.X, event.Y

form.MouseMove += onMouseMove
```

We turn this into something more interesting by building on our example with a label. When the mouse moves over the label, the color of the label text and background will change. You'll make the color random by using the Random class from the System namespace and the FromArgb method of the Color structure (listing 3.2).

Listing 3.2 Wiring up the MouseMove event on a Label control

```
import clr
clr.AddReference('System.Windows.Forms')
clr.AddReference('System.Drawing')
from System.Windows.Forms import (
   Application, Form,
   FormBorderStyle, Label
)
from System.Drawing import (
   Color, Font, FontStyle, Point
)
from System import Random

random = Random()

form = Form()
form.Text = "Hello World"
form.FormBorderStyle = FormBorderStyle.Fixed3D
form.Height = 150

newFont = Font("Verdana", 16,
   FontStyle.Bold | FontStyle.Italic)

label = Label()
label.AutoSize = True
label.Text = "My Hello World Label"
label.Font = newFont
label.BackColor = Color.Aquamarine
label.ForeColor = Color.DarkMagenta
label.Location = Point(10, 50)

form.Controls.Add(label)
```

```
def GetNewColor():
    red = random.Next(256)
    green = random.Next(256)
    blue = random.Next(256)
    return Color.FromArgb(red, green, blue)

def ChangeColor(sender, event):
    print 'X:', event.X, 'Y:', event.Y
    sender.BackColor = GetNewColor()
    sender.ForeColor = GetNewColor()

label.MouseMove += ChangeColor

Application.Run(form)
```

GetNewColor returns random color

Defines event handler function

Changes label colors

Figure 3.7 gives you an idea of what to expect when you run this code. You should see a form and label looking similar to the previous example that introduced the `Label` class. When you move the mouse over the label, the `MouseMove` event is raised, and the handler is called. The label text and background colors should change to random colors, and the mouse location is printed.



Figure 3.7 Hello World `form` with a `label` responding to `MouseMove` events

You might have noticed that the values for `X` and `Y`, the mouse coordinates, are low. The coordinates are the position of the mouse *within* the label, called the *client area* of the label. The top left of the label is position 0,0. Controls have methods to convert locations within their client area to screen coordinates[14] (and vice versa); but, when specifying locations within a control, you always use client coordinates. When you position controls within a container, you don't need to know where they are on the screen or their parent controls.

You also use a new class, `Random`, from the `System` namespace. The IronPython engine already has a reference to the `System` assembly, so you don't need to explicitly add one. The `Random` class is an easy-to-use, pseudo-random number generator. You create an instance of `Random` at the module level; the `GetNewColor` function uses it to fetch three integers from 0–255 and return a new color.

We've gone through the techniques to look up classes in the MSDN documentation (almost in excruciating detail for the early examples). If you want to know more about the `Random` class, we'll leave it as an exercise for you to find the documentation, both for the class and the `Next` method.

The examples so far have passed form instances directly to `Application.Run`. A much better way of organizing your code is to create a subclass of `Form` and do your setup in the constructor. This also makes a great example of subclassing a .NET class in IronPython.

[14] `PointToScreen` and `PointToClient` methods.

3.4 Subclassing .NET types

Creating IronPython classes that inherit from .NET types works fine. In listing 3.3, you inherit from `Form` to better organize the code.

Listing 3.3 A form configured in the constructor of a `Form` subclass

```
import clr
clr.AddReference('System.Windows.Forms')
clr.AddReference('System.Drawing')
from System.Windows.Forms import (
   Application, Form,
   FormBorderStyle, Label
)
from System.Drawing import (
   Color, Font, FontStyle, Point
)

class MainForm(Form):                       <- Class declaration inheriting from Form

   def __init__(self):
      self.Text = "Hello World"                        <- Properties are
      self.FormBorderStyle = FormBorderStyle.Fixed3D      configured on self
      self.Height = 150

      newFont = Font("Verdana", 16,
         FontStyle.Bold | FontStyle.Italic)

      label = Label()
      label.AutoSize = True
      label.Text = "My Hello World Label"
      label.Font = newFont
      label.BackColor = Color.Aquamarine
      label.ForeColor = Color.DarkMagenta
      label.Location = Point(10, 50)

      self.Controls.Add(label)

mainForm = MainForm()
Application.Run(mainForm)
```

This code has exactly the same result as the first Hello World example with the label. The only difference is that the form is configured inside the `__init__` method (the constructor). `__init__` is called when a new instance of `MainForm` is created; inside the body of this method, the variable `self` refers to the new instance. Because `MainForm` is a subclass of `Form`, it has all the same properties and is configured in the same way you saw earlier.

By now, you can see how easy it is to work with the .NET framework from IronPython. If you're a Python programmer, you should note that the different .NET types merely behave as objects with attributes and methods, and you have a (large) new set of libraries to work with. If you're a .NET programmer, then all your knowledge of .NET is relevant to IronPython. Now that we've covered the basic principles, it's time to start on the meat!

3.5 Summary

.NET objects come in various types, which are part of a type system called the Common Type System.[15] Most of the types in this system fall into one of the following categories:

- Built-in types
- Pointers
- Classes
- Structures
- Interfaces
- Enumerations
- Delegates

We've encountered or discussed all of these types, except for pointers (and we need to discuss interfaces a bit more deeply). Pointers[16] are mainly used for interacting with unmanaged resources, and don't turn up in most applications. You've also seen how easy it is to use .NET objects from within IronPython; it can be simpler and require less lines of code than using them with a language like C#!

In the next couple of chapters, you'll build an IronPython application that takes advantage of a wider selection of the huge number of framework classes available to you. As the application gets larger, a good structure will be essential for keeping the code readable and easy to maintain and extend. We'll be exploring good Python practice, and growing your understanding of IronPython and .NET, by using some common design patterns.

[15] See http://msdn2.microsoft.com/en-us/library/2hf02550.aspx.
[16] You can find a good reference on pointers at http://www.codeproject.com/KB/dotnet/pointers.aspx.

Part 2

Core development techniques

In part 1, we went over the basics of Python and interacting with the .NET framework from IronPython. In this part, we explore the core development techniques needed for writing applications with IronPython.

Essential to writing large programs for any platform is well-structured code. Without clear structure, it becomes impossible to keep an overview of what parts of the program perform which functions. More importantly, a well-structured program will be much easier to extend and more easily understood by others reading the code.

At the heart of the high-level techniques we use are design patterns. The term design patterns was popularized by a book of the same name, written in 1995 by four authors now immortalized as the Gang of Four. Design patterns formulate common tasks in programming and provide extensible object-oriented solutions. By recognizing places in your programs that are well suited to the use of these patterns, you can take advantage of well-tried approaches that provide a blueprint for aspects of your program structure. In the following chapters, we look at how to apply a few design patterns in IronPython and dive further into both Python and the .NET framework.

Writing an application and design patterns with IronPython

This chapter covers

- Duck typing in Python
- The Model-View-Controller pattern
- The Command pattern
- Integrating new commands into MultiDoc

Python as a language goes out of its way to encourage a good clear programming style. It certainly doesn't enforce it, though; you're just as free to write obscure and unmaintainable code with Python as you are with any other programming language. When talking to programmers who haven't used Python, I'm sometimes surprised by their perception that it's harder to write well-structured programs with Python because it's a dynamically typed language. In fact, by making it easy to express your intentions and reducing the amount of code you need to write, Python makes large projects *more* maintainable.

In this chapter, as well as learning more about IronPython, you'll also explore some Python programming best practices. This exploration includes getting deeper inside Python by using some of Python's magic methods. You'll also create a small application to tackle some common programming tasks, using several .NET namespaces that you haven't yet seen.

We start with a look at data modeling in Python.

4.1 Data modeling and duck typing

Computer science has many difficult problems, but one problem central to many applications is data modeling. Data modeling means structuring and organizing data, both for a computer to be able to work with the data and to make it possible to coherently present the data to humans. This means finding the right way of representing the data. The structures you choose must be a good fit, and you must provide the right API for accessing and modifying the data.

The built-in datatypes, such as the list and the dictionary, provide straightforward ways of accessing data. They even have syntax built into the language for doing so. Data modeling in Python is often done by creating structures that reuse the same way of accessing the underlying information. In .NET languages, you might provide data access by implementing interfaces; Python achieves the same thing using protocols.

4.1.1 Python and protocols

The traditional .NET languages are all statically typed. To call methods on an object, the compiler needs to know what type it is and that the type has the required method. Certain properties of objects, such as being able to iterate over them and index them, are supported in .NET through interfaces. If a class implements an interface, then the compiler knows that it supports certain types of operations. The .NET interfaces that allow indexing by position or by key are `IList` and `IDictionary`.

Instead of interfaces, Python uses protocols. The Python equivalent of `IList` is the sequence protocol; the equivalent of `IDictionary` is the mapping protocol. An object can support either of these protocols by implementing the `__getitem__` and/or the `__setitem__` and `__delitem__` methods (table 4.1). Python uses the same methods for both protocols.

This table shows how using indexing to set/fetch or delete an item translates to Python method calls. Instead of being called directly, Python calls the indexing

Table 4.1 The Python sequence and mapping protocol magic methods

Python syntax	Translates to	Operation
`x = something[key]`	`x = something.__getitem__(key)`	Fetching an item by key or index
`something[key] = x`	`something.__setitem__(key, x)`	Setting an item by key or index
`del something[key]`	`something.__delitem__(key)`	Deleting an item

methods for you when it encounters an indexing operation. These method names start and end with double underscores—which marks them as *magic methods*. There are many more protocols for other operations such as comparison, numerical operations, and iteration. These operations also have corresponding magic methods (and many have equivalent .NET interfaces).

Let's look at how protocols work in practice with the sequence and mapping protocols.

4.1.2 Duck typing in action

These protocols are duck typing in action. An indexing operation will succeed if the indexed object supports the appropriate magic method, irrespective of what type it is.

> **Duck typing: a short history**
>
> The term *duck typing* gets its name from the duck test (attributed to James Whitcomb Riley):
>
> > *When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck.*
>
> Wikipedia attributes the first known use of the term duck typing to Alex Martelli in a usenet post to the comp.lang.python newsgroup in 2000.[1]
>
> The essence of this concept is that valid uses of an object should be determined by what operations it supports rather than what type it is.

Duck typing can be extremely useful. If you have an API that takes a mapping type object and sets or fetches entries from the object, you can pass in any object that implements the mapping protocol. Protocols are a dynamically typed versions of interfaces. They allow you to create data structures as classes, implementing custom behavior when data is set or fetched. You can also provide additional methods to access or modify the data beyond simple access.

Listing 4.1 is an example class that prints to the console whenever you fetch, set, or delete entries. It uses a dictionary as the underlying data store. In practical terms, you may want to do something more useful than printing, but this is only an example!

Listing 4.1 A custom mapping type

```
lass ExampleMappingType(object):
   def __init__(self):
      self._dataStore = {}

   def __getitem__(self, key):
      value = self._dataStore[key]
      print 'Fetching: %s, Value is: %s' % (key, value)
```

[1] See http://en.wikipedia.org/wiki/Duck_Typing.

```
        return value

    def __setitem__(self, key, value):
        print 'Setting: %s to %s' % (key, value)
        self._dataStore[key] = value

    def __delitem__(self, key):
        print 'Deleting:', key
        del self._dataStore[key]

>>> mapping = ExampleMappingType()
>>> mapping['key1'] = 'value1'
Setting: key1 to value1
>>> x = mapping['key1']
Fetching: key1, Value is: value1
>>> print x
value1
>>> del mapping['key1']
Deleting: key1
```

Any Python code that expects a Python dictionary, but only performs indexing operations on it, can use `ExampleMappingType`. If the code does more than indexing operations—for example, calling dictionary methods such as `keys` or `items`—you could implement these as well. If you had wanted to implement a sequence type instead of a mapping type, then the magic methods implemented here would have worked with numbers instead of keys for the index.

We've completed our look at data modeling and protocols. So far it has been quite abstract. Let's see if we can put this knowledge, along with everything else you've learned about Python, into use with some more substantial design patterns. The first one is the Model-View-Controller pattern, and it will form the basis of our running example.

4.2 Model-View-Controller in IronPython

A good structure for applications with a user interface and an underlying data model is the Model-View-Controller pattern, often referred to as MVC. The basic structure is shown in figure 4.1.

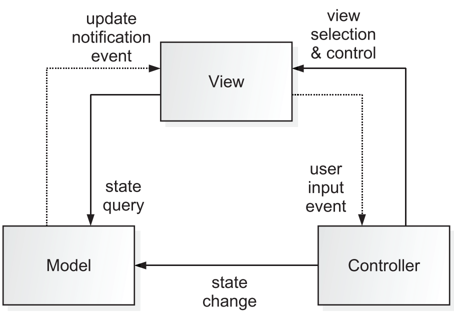


Figure 4.1 The basic structure of Model-View-Controller. The Controller mediates between the View and the Data Model, which remain separated.

The layer of your application that interacts with the user is the view. In a desktop application, this is the GUI code.

The model is an object-oriented representation of your data. Its structure depends on what kind of data your application is dealing with. Although the view may query the model, the model knows nothing about the view. Changes to the model are made through the controller. The advantage of this structure is that your data model is then entirely independent of the view layer, keeping a good separation of code. Additionally,

if you turn your program into a web application or rewrite with a different type of user interface, then the data model can remain unchanged.

The controller mediates between the view and the model. It uses the public API of the model to change data in response to user input. It also supplies the view with the data to be presented to the user.

The view may subscribe to events on the model to be kept aware of internal state changes; it may also directly query the model, but changes go through the controller.

Model-View-Controller will be the core pattern in our running example.

4.2.1 Introducing the running example

In the next few chapters, you'll build an IronPython application. Our running example, called MultiDoc, is a Windows Forms desktop application for editing multipage documents and saving them as a single file. Using Windows Forms not only makes for good visual examples, but it should also be possible to refactor the application to use an alternative user interface if you build the application carefully.

> **Code examples**
>
> As we build on the running example, or show other code samples through the book, only the new or changed portions of code will be shown. This keeps the listings shorter and means we can fit more examples in the book!
>
> To make the examples runnable, you'll sometimes need to add imports or parts of the code that we've already covered. All the examples are available for download in full from the *IronPython in Action* website.[2]

As we develop MultiDoc, you'll learn some further aspects of Python and use a wider range of .NET components. More importantly, we focus on good design practice and effective use of common Python idioms. Our editor should perform the following tasks:

- Allow the user to edit and create text documents
- Divide each document into multiple pages, which are edited in a separate tab of a multitabbed control
- Allow the user to add or delete pages
- Allow the document to be saved as a single file and then reloaded

More features may be a natural fit for MultiDoc, but this is the initial specification; let's see where it goes.

You can initially provide a simple GUI and then gradually (incrementally) add more features. The advantage of this approach is that, once you've created the first simple version, you can maintain a working application at every stage. New features

[2] See http://www.ironpythoninaction.com.

can be added and tested; if necessary, you can refactor the structure as the program grows. To continue with the construction analogy: you'll first build a small shed; then you'll refactor it into a bungalow, adding a new floor and some modern conveniences; and finally you'll have a three-bedroom house. I'm not sure we'll have time to get to the mansion stage, but you ought to end up with an attractive little piece of real estate (and fortunately, refactoring code is less work than remodeling houses).

The user interface is the view layer in the Model-View-Controller pattern, and this is where we start creating MultiDoc.

4.2.2 The view layer: creating a user interface

The specification says that the user edits documents using an interface with multiple tabs, one tab per page. The Windows Forms control for presenting a multitabbed interface is the `TabControl`, a container control. Individual pages within the `TabControl` are represented with `TabPage` controls. Let's create a form with a `TabControl` in it to establish the base of the view layer of the application; you'll build on this as you add new features.

The `TabControl` will form the main component in the MultiDoc user interface. Windows Forms handles drawing the tabs and switching between the pages; but, if you need to, you can implement custom behavior when the user selects a new tab. In this first prototype of MultiDoc, you'll add a method to create a new `TabPage`, which can contain the `TextBox`—the basic Windows Forms text editing control. To keep things simple, you'll start with a single `TabPage` (listing 4.2).

Listing 4.2 `TabControl` with single `TabPage` and multiline `TextBox`

```
import clr
clr.AddReference('System.Windows.Forms')
clr.AddReference('System.Drawing')

from System.Drawing import Size
from System.Windows.Forms import (
   Application, DockStyle, Form, ScrollBars,
   TabAlignment, TabControl, TabPage, TextBox
)

class MainForm(Form):
   def __init__(self):
      self.Text = 'MultiDoc Editor'
      self.MinimumSize = Size(150, 150)

      self.tabControl = TabControl()
      self.tabControl.Dock = DockStyle.Fill          <- Fills form with tab control
      self.tabControl.Alignment = (
         TabAlignment.Bottom)                        <- Puts tabs at bottom
      self.Controls.Add(self.tabControl)

      self.addTabPage("New Page", '')

   def addTabPage(self, label, text):
      tabPage = TabPage()
      tabPage.Text = label

      textBox = TextBox()
```

```
        textBox.Multiline = True
        textBox.Dock = DockStyle.Fill
        textBox.ScrollBars = ScrollBars.Vertical
        textBox.AcceptsReturn = True
        textBox.AcceptsTab = True
        textBox.WordWrap = True
        textBox.Text = text

        tabPage.Controls.Add(textBox)

        self.tabControl.TabPages.Add(tabPage)

Application.EnableVisualStyles()
Application.Run(MainForm())
```

Enables vertical scrollbars

Enables XP themes

The code in this listing creates the embryonic MultiDoc editor. Although it uses controls that you haven't specifically seen yet, they're configured in exactly the same way as the ones that you've already worked with. If you run this code, you should get a form with a tab control, containing a single tab page filled with a text box, as shown in figure 4.2.



Figure 4.2 The first cut of the MultiDoc editor: a single tab page with a multiline text box

NOTE You can set two properties on a control to specify the way the control is laid out in the parent control: `Dock` and `Anchor`. Both are configured by setting an enumeration (`DockStyle` and `AnchorStyles`), and control the way that the child control is resized when the parent is resized.

Figure 4.3 shows the same code running under Mono on Mac OS X (Mono 1.9,[3] IronPython 1.1, and Mac OS X 10.5.2 if you need to know these things). All the code in the running example should run fine on Mono with little to no changes.

There's almost no logic in listing 4.2; *most* of the code is concerned with configuring the controls you use. (And you could do a lot more configuration.) One new thing in this listing is the setting of the `MinimumSize` property to specify a minimum size for the main form. By default, you can resize a form down to almost nothing—which looks a bit odd; this setting solves this problem. `MinimumSize` takes a `Size` structure, which is constructed from two integers.

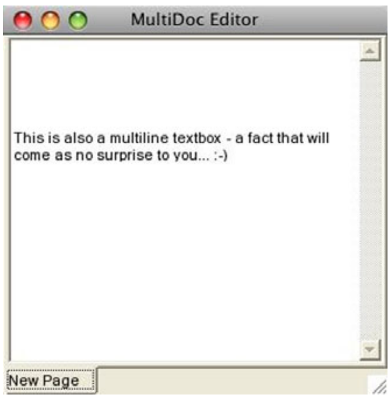


Figure 4.3 MultiDoc running under Mono on the Apple Mac

[3] Later versions of Mono, 2.0 onwards, have several important bugs fixed for the Mac, so it is worth using the latest version if possible.

Before starting the event loop, you also call `Application.EnableVisualStyles`. This enables styles so that the application will use any theme that the user may have set under Windows.

The `TabControl` is positioned within the form, using the `Dock` property and the `DockStyle` enumeration. `DockStyle.Fill` is used to tell the form to fill all the available space with the `TabControl`. You do the same thing with the `TextBox`. As you add more features to the GUI, you'll be using different values for `DockStyle`. For example, you'll probably position a menu or toolbar using `DockStyle.Top`.

The new tab page is created by the `addTabPage` method, called from the constructor. `addTabPage` has to do a lot of configuration of the text box to make it usable as a minimal editor. You want to be able to use the tab key to enter tabs rather than moving the focus; you'd like vertical scrollbars if there's more text in the text box than fits in the available space; and so on. This is all fairly straightforward.

The code written so far is the skeleton of a user interface. It doesn't deal at all with how you'll represent the user data or how the user interface will interact with the data; for this you need to create a data model.

4.2.3 A data model

To keep MultiDoc easy to maintain, you should define standard ways of accessing and changing the document. This will be useful for saving documents, loading new documents, switching between individual pages in the document, and so on.

The documents in our example are intended to contain multiple pages; a logical data model would be a document containing pages, with both the document and the pages represented in classes. Pages are stored sequentially, so it makes sense to make the document a sequence-type container for pages.

Pages need to store the text in them; plus, they have a title displayed on the tab. As this is the only attribute they have (for the moment), you can represent them with a simple class (see listing 4.3).

Listing 4.3 Data model with `Document` and `Page` classes

```
class Document(object):
   def __init__(self, fileName=None):
      self.fileName = fileName
      self.pages = []
      self.addPage()                      # Creates one empty page

   def addPage(self, title='New Page'):
      page = Page(title)                  # Creates new page
      self.pages.append(page)

   def __getitem__(self, index):
      return self.pages[index]

   def __setitem__(self, index, page):
      self.pages[index] = page

   def __delitem__(self, index):
```

```
        del self.pages[index]

class Page(object):                          <- Class that represents pages
    def __init__(self, title):
        self.title = title
        self.text = ''
```

This listing shows the code to implement a simple data model with `Document` and `Page` classes. For the moment, all the code will be kept in a single source file, but the listing only shows the additional code for the `Document` and `Page` classes. By default, the `Document` class creates one empty page when it's instantiated. Thanks to the compactness of Python, there's very little boilerplate, and every line does something of note. We've left it up to you to deduce the meaning of most of the code.

To hook this into the presentation layer (the view) that you've already created, you need a controller.

4.2.4 A controller class

The main element in the MultiDoc user interface is the `TabControl`. It has `TabPages`, which map to the pages that the document holds.

The controller is responsible for setting the data on the model, and for keeping the view and the model synchronized. In this way, the code is neatly separated between the view, which manages the GUI layout, and the controller, which responds to the user input.

As the application grows, it may need several controller classes for the major GUI elements; the first will be a `TabController`. Because this will be responsible for adding new pages, the `addTabPage` method can be moved into the controller. To do this, you'll need a reference to both the `TabControl` instance on the `MainForm`, and the current document. When `MainForm` is instantiated, it should create a document and then instantiate the `TabController`, passing in both the `TabControl` and the new document.

The `TabController` can then iterate over all the pages in the document and create a `TabPage` for each one. When you come to loading documents with multiple pages already in them, creating the appropriate `TabPages` in the view will then happen automatically (listing 4.4).

Listing 4.4 `TabController` with changed `MainForm` class

```
class TabController(object):
    def __init__(self, tabControl, document):
        self.tabControl = tabControl
        self.document = document

        for page in document:                              <- Creates tab page for each page
            self.addTabPage(page.title, page.text)

        self.index = self.tabControl.SelectedIndex
        if self.index == -1:
            self.index = self.tabControl.SelectedIndex = 0          <- Selects first tab if necessary
        self.tabControl.SelectedIndexChanged += self.maintainIndex  <- Maintains index when tab changes
```

```
    def addTabPage(self, label, text):            ◁ Moved from MainForm
        tabPage = TabPage()
        tabPage.Text = label

        textBox = TextBox()
        textBox.Multiline = True
        textBox.Dock = DockStyle.Fill
        textBox.ScrollBars = ScrollBars.Vertical
        textBox.AcceptsReturn = True
        textBox.AcceptsTab = True
        textBox.WordWrap = True
        textBox.Text = text

        tabPage.Controls.Add(textBox)

        self.tabControl.TabPages.Add(tabPage)

    def maintainIndex(self, sender, event):           Updates model
        self.updateDocument()                       ◁ with current text
        self.index = self.tabControl.SelectedIndex

    def updateDocument(self):
        index = self.index
        tabPage =  self.tabControl.TabPages[index]       Text box from
        textBox = tabPage.Controls[0]                 ◁ current tab page
        self.document[index].text = textBox.Text

class MainForm(Form):
    def __init__(self):
        Form.__init__(self)
        self.Text = 'MultiDoc Editor'
        self.MinimumSize = Size(150, 150)

        self.tabControl = TabControl()
        self.tabControl.Dock = DockStyle.Fill
        self.tabControl.Alignment = TabAlignment.Bottom
        self.Controls.Add(self.tabControl)

        self.document = Document()
        self.tabController = TabController(self.tabControl, self.document)

Application.EnableVisualStyles()
Application.Run(MainForm())
```

This listing shows the new `MainForm` creating a document and initializing the tab controller. The document creates one page by default, and the tab controller then automatically creates a tab page.

NOTE The tab controller creates a tab page for each page in the document by iterating over them. You haven't specifically implemented the protocol methods for iteration (`__iter__` and `next`), but you get a default implementation for free along with the sequence protocol method `__getitem__`.

The tab controller has the `updateDocument` method for keeping the document updated with changes from the user interface. This method needs to be called before saving the document, or every time the active tab is changed.

Keeping the document updated means you need to keep track of the currently selected page—which is done with the `self.index` instance variable. At the end of the `TabController` constructor, you set the index to the currently selected tab and register the `maintainIndex` method to be called whenever the selected tab changes. Whenever the selected tab changes, the `SelectedIndexChanged` event fires.

When `maintainIndex` is called, the selected index has *already* changed. It calls `updateDocument`, which fetches the text from the tab that was *previously* selected (by using the `index` variable). It puts this text into the equivalent page on the document; each tab has a corresponding page in the document, so the index of the tab also acts as the index of the page on the document. After updating the model, `maintainIndex` then stores the index of the newly selected tab; the next time the user switches tabs, this one will be synced to the model. The point of all this is so that, when you need to save the document, you know that you only need to sync the currently selected page, and then you can save the document.

There's a problem, though. When the form hasn't yet been shown, the `SelectedIndex` will be -1. When the form is shown, the selected index will then be set, but the `SelectedIndexChanged` event is *not* fired. Adding some extra code in the constructor allows you to get around this; if `SelectedIndex` returns -1, then you explicitly select the first tab.

You've now created the core structure for the application; but, if you run this new version of MultiDoc, it looks identical to the previous version. So far you've refactored the code to provide a better structure. You've also learned a bit more about Python and dived into using design patterns. In the coming sections we continue developing MultiDoc, but use more .NET classes that you haven't seen yet.

The next step in our exploration is extending MultiDoc using the command pattern.

4.3 The command pattern

A command pattern is *where a request is encapsulated as an object*.[4] In practice, this means that you represent each new command as a class, and provide a common interface for command classes to perform actions. The command pattern makes it extremely easy to add new commands.

We want to add the capability to MultiDoc to save documents that are being edited (currently only a single page). Traditionally, applications offer the following two types of save commands:

- A Save command, which uses the current filename without prompting for one if the document has been saved before
- A Save As command, which always prompts for a filename

Let's start with the Save command. If we're careful, it should be easy to reuse most of the code from the Save command in the Save As command.

---

[4] Duncan Booth, *Patterns in Python*, http://www.suttoncourtenay.org.uk/duncan/accu/pythonpatterns.html.

To save files, the `SaveFileDialog` class will be an integral part. This class is the standard file dialog box used by almost every Windows application.

4.3.1 The SaveFileDialog

The `SaveFileDialog` presents the user with an explorer window so that a filename can be selected to save a file with. The dialog box is shown modally—it blocks the application until the user has selected a file.

To use this dialog box, you need to configure the filter, which determines which file types are shown in the explorer, and the title. You may also want to configure which directory the browser opens in and if any initial filename is to appear there. As with other controls you've used, this is all done by setting simple properties.

The dialog box is displayed by calling the `ShowDialog` method. This method blocks until the user has selected a filename or hit the cancel button, but you need some way of telling which of these two actions the user has performed. `ShowDialog` returns either `DialogResult.OK` or `DialogResult.Cancel`, depending on whether the user selects a file or cancels. `DialogResult` is an enumeration, which .NET is so fond of. If this were a native Python GUI toolkit, then calling `ShowDialog` would probably return only `True` or `False`!

> **Path or os.path**
>
> Manipulating file paths is such a common operation that both the .NET framework and the Python standard library provide functions for working with them.
>
> In MultiDoc, we use static methods on the .NET class `System.IO.Path` for obtaining the filename and directory name from a path.
>
> The Python module `os.path` provides similar functionality. `os.path.split` takes a path and splits it into the following two parts:
>
> ```
> import os
> directory, filename = os.path.split(filePath)
> ```

Listing 4.5 shows an interactive interpreter session which creates and configures a Save File dialog box and displays the result. It takes an existing file path and uses the `Path` class from `System.IO`[5] to extract the file and directory name from the path.

Listing 4.5 Configuring and displaying `SaveFileDialog`

```
>>> import clr
>>> clr.AddReference('System.Windows.Forms')
>>> from System.Windows.Forms import SaveFileDialog, DialogResult
>>> from System.IO import Path
```

[5] `IO` is a namespace in the `System` assembly. You don't need to explicitly add a reference to the `System` assembly because the IronPython engine uses it.

```
>>> filepath = r'C:\filename.txt'
>>> filter = 'Text files (*.txt)|*.txt|All files (*.*)|*.*'
>>> directory = Path.GetDirectoryName(filepath)
>>> name = Path.GetFileName(filepath)
>>> dialog = SaveFileDialog()          <—— Creates new dialog box
>>> dialog.FileName = name
>>> dialog.InitialDirectory = directory
>>> dialog.Filter = filter
>>> dialog.Title = 'Just an example'
>>> result = dialog.ShowDialog()       <—— Displays dialog box
>>> result == DialogResult.OK
True
>>> dialog.FileName
'C:\\Some Path\\Some File.txt'
```

Figure 4.4 shows the result of running the code from this listing; a Save File dialog box is displayed with all the properties that you've set. If you select a file with the dialog box, then `DialogResult.OK` will be returned, and the `FileName` attribute set appropriately. If you hit the cancel button instead, then `DialogResult.Cancel` is returned.

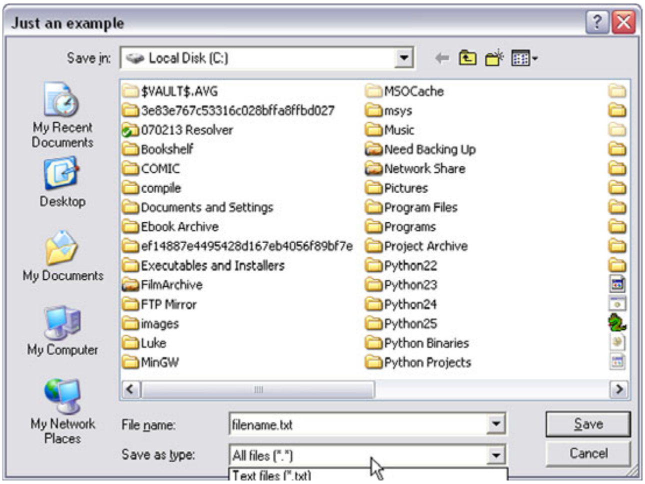


Figure 4.4
A `SaveFileDialog` configured from the interactive interpreter and displayed with `ShowDialog`

You're one step closer toward being able to create the Save command. Now that you've ascertained that the user wants to save the file, you need to write out the file with the chosen filename.

4.3.2 Writing files: the .NET and Python ways

Initially, you're going to write out the contents of the text box, from the `Text` property, as a text file. Strings on .NET are stored using Unicode, and IronPython strings are no exception. Unicode is an abstract representation of text; when you write the text to the filesystem, an encoding must be used. The encoding is a concrete way of representing text by assigning digits to each character. .NET *can* choose a default encoding for you, but you need to be aware that this is happening.

> **Encodings**
>
> Hundreds of different encodings are in use across the world, and reading text without knowing the encoding can cause characters to be read incorrectly. Luckily, .NET handles this most of the time.[6]
>
> The most common encoding, and also one of the most limited, is ASCII, which uses 7 bits to represent all the characters in the English alphabet.
>
> A common alternative to ASCII is UTF-8, which uses the same numbers as the ASCII characters but can also use extra bytes to represent every character in the Unicode standard.

.NET provides several different ways of writing text files, built on top of the `FileStream` class. `FileStreams` read or write, with either text or binary data, to files on the filesystem. .NET also provides the `StreamWriter` class, an abstraction layer on top of the `FileStream` for the easy writing of text files. The corresponding class for reading text files is the `StreamReader`.

To initialize a `StreamWriter`, you can either pass in a filename (as a string), or a ready-created `FileStream` instance. You can also specify an encoding; but, if you omit this, then the default encoding will be used. When developing with Python, the default encoding is ASCII; when using .NET framework classes, the default encoding will usually be UTF-8. Using UTF-8 is likely to result in many less Unicode headaches.

The following segment of code creates a `StreamWriter` and writes a string to a file:

```
>>> from System.IO import StreamWriter
>>> writer = StreamWriter('filename.txt')
>>> writer.Write('some text')
>>> writer.Close()
```

TIP The class `System.IO.File` also has some convenient static methods for reading and writing files. `File.ReadAllText` and `File.WriteAllText` (and the corresponding methods for reading and writing bytes) mean that reading or writing files can be a one-liner. They don't give as fine a degree of control, but are often all that you need.

If you want to open the file in any mode other than write (for example, to append to a file), you'll need to create the `FileStream` and specify a `FileAccess` mode. Python also provides standard ways of working with files. It uses a single built-in function, `open`, for both reading and writing files. You specify the mode with one of the following strings:

- *r*—For reading text files
- *w*—For writing text files

[6] It uses `System.Text.Encoding.Default`, which depends on the user's regional and language settings. On a UK English Windows machine, it's likely to be ISO 8859-1.

- *rb*—For reading binary files
- *wb*—For writing binary files

In Python, the string class is used for representing both text and binary data, so file handles (returned by `open`) have `read` and `write` methods that always take or return strings.

```
>>> handle = open('filename.txt', 'w')
>>> handle.write('some text')
>>> handle.close()
>>> handle = open('filename.txt', 'r')
>>> text = handle.read()
>>> handle.close()
>>> print text
some text
```

If your code is making heavy use of .NET classes, then it may make sense to use the .NET way. If you want your code to run on both CPython and IronPython, then you should use the Python patterns for working with files.

The code segments shown so far for writing files will work fine, as long as nothing goes wrong. Unfortunately, many things can go wrong when writing to external devices; the device may be read-only, or the drive may be full. If this happens, you have to be able to handle it *and* report it to the user.

4.3.3 Handling exceptions and the system message box

We've already covered handling exceptions in Python; now you get a chance to use it. If saving a file fails, for whatever reason, you want to catch the exception and inform the user of the problem.

This is a good example of where the creators of IronPython have done an excellent job of making the boundary between the .NET and the Python world seamless. The exception raised by .NET for this kind of error is an `IOException`. IronPython converts this into a Python exception. The corresponding Python exception type is called `IOError`. In these examples, we're going to use features from both the .NET platform *and* the Python language. As you saw in the previous section, this can be done using standard Python patterns or with the .NET classes, a technique that should feel familiar to .NET programmers. This is the essence of IronPython at work: the flexibility of Python combined with the power and breadth of .NET. Using IronPython from the interactive interpreter, as you've been doing, illustrates how useful IronPython can be for experimenting with .NET classes, even if you're coding in straight C#. Table 4.2 shows a mapping of some common Python exceptions to their .NET equivalents. Not all Python exceptions have .NET equivalents; for these cases, custom exceptions are defined inside IronPython[7] (such as `RuntimeError`, `NameError`, `SyntaxError`, and `ImportError`).

---

[7] A reference and discussion of custom exceptions in IronPython 1 can be found at http://www.codeplex.com/IronPython/Wiki/View.aspx?title=Exception%20Model.

Table 4.2 Python exceptions mapping to .NET exceptions

Python exception	.NET exception
Exception	System.Exception
StandardError	SystemException
IOError	IOException
UnicodeEncodeError	EncoderFallbackException
UnicodeDecodeError	DecoderFallbackException
MemoryError	OutOfMemoryException
Warning	WarningException
StopIteration	InvalidOperationException subtype
WindowsError	Win32Exception
EOFError	EndOfStreamException
NotImplementedError	NotImplementedException
AttributeError	MissingMemberException
IndexError	IndexOutOfRangeException
KeyError	System.Collections.Generic.KeyNotFoundException
ArithmeticError	ArithmeticException
OverflowError	OverflowException
ZeroDivisionError	DivideByZeroException
TypeError	ArgumentTypeException

When you catch the exception, the instance of `IOError` raised will carry a message telling you what went wrong. You can turn the exception instance into a string by using the built in `str` function. The following code segment tries to save a file with a bogus filename—unless you have this file on drive z, that is! It catches the ensuing error and reports it by printing the error message:

```
>>> from System.IO import StreamWriter
>>> filename = 'z:\\NonExistentFile.txt'
>>> try:
...     writer = StreamWriter(filename)
... except IOError, e:
...     print 'Something went wrong.'
...     print 'The error was:\r\n%s' % str(e)
...
Something went wrong.
The error was:
Could not find a part of the path 'z:\NonExistentFile.txt'
```

String interpolation

This segment of code uses string interpolation to format the error message. The placeholder %s is used inside the string, and then the % operator (the modulo operator) is used to interpolate values into the string.

To interpolate multiple values, follow the % operator with a tuple. There are quite a few different placeholders: %d for decimal numbers, %f for floating point numbers (with a variety of options for how they're formatted), and so on.

For example:

```
myString = 'my name is %s, my age is %d. I am holding %r' % ('Michael', 32, SomeObject)
```

But MultiDoc is a GUI application, and you don't want to rely on the console for reporting problems. The standard way for informing the user of this sort of problem is using the system message box.

As with the other GUI elements we've been working with, you can get access to the system message box through the Windows Forms namespace. Instead of instantiating a message box, you call the static method `Show` on the `MessageBox` class.

The `Show` method has a plethora of different overloads to control the way you display the message box. You can configure the message, the title, choose an icon, and specify the number and type of buttons on the message box. The following code segment displays a message box with a title and message, with an icon and an OK button. You use a couple of enumerations to specify the buttons and the icon.

```
>>> from System.Windows.Forms import MessageBox, MessageBoxButtons,
      MessageBoxIcon
>>> message = 'Hello from a MessageBox'
>>> title = 'A MessageBox'
>>> MessageBox.Show(message, title,
... MessageBoxButtons.OK, MessageBoxIcon.Asterisk)
<System.Windows.Forms.DialogResult object at 0x… [OK]>
>>>
```



Figure 4.5 A system message box created from an interactive interpreter session

If you enter the code from this segment into an interactive interpreter session, then you should see the message box shown in figure 4.5.

The *i* in a bubble is the icon specified with `MessageBoxIcon.Asterisk`. You can pick from a veritable smorgasbord of icons with this enumeration. You can also use Yes, No, and Cancel buttons instead of the OK button (or various other button combinations) by changing the value from the `MessageBoxButtons` enumeration. The call to `Show` returns a `DialogResult`, depending on the button used to exit the message box. You can find all the possible values for these enumerations in the MSDN documentation.

You're now ready to implement the Save command. You know how to ask the user to choose a filename, how to write out the file, and how to handle and report any problems that occur. Let's put all this together.

4.3.4 The SaveCommand

We intend to implement two save-like commands, the Save command and the Save As command. Save As will be similar to the Save command, so let's ensure that as much of the code as possible from the Save command can be reused. Figure 4.6 shows the sequence of actions for the Save command.

The Save As command will be very similar to the sequence shown in this figure, except that it will still present the file dialog box to the user even if the document already has a filename associated with it.

The command will need access to the text from the document, which will need to be updated. The easiest way for the command to update the document is to give it a reference to the tab controller (listing 4.6).

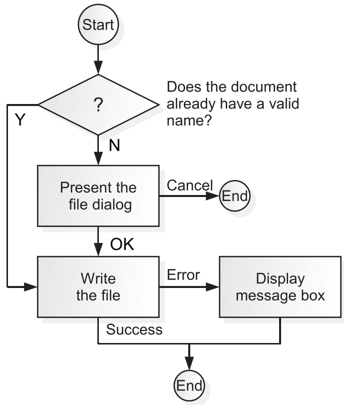


Figure 4.6 A flow chart for the actions of the Save command

Listing 4.6 `SaveCommand`, which handles necessary user actions and writes file to disk

```
from System.IO import Directory, Path, StreamWriter
filter = 'Text files (*.txt)|*.txt|All files (*.*)|*.*'

class SaveCommand(object):

   title = "Save Document"

   def __init__(self, document, tabController):
      self.document = document
      self.tabController = tabController
      self.saveDialog = SaveFileDialog()             ❶
      self.saveDialog.Filter = filter
      self.saveDialog.Title = self.title

   def execute(self):                           ❷
      fileName = self.document.fileName
      text = self.getText()

      directory = Path.GetDirectoryName(fileName)
      directoryExists = Directory.Exists(directory)         ❸
      if fileName is None or not directoryExists:
         self.promptAndSave(text)
      else:
         self.saveFile(fileName, text)

   def getText(self):
      self.tabController.updateDocument()
      return self.document[0].text

   def promptAndSave(self, text):
```

```
        saveDialog = self.saveDialog
        if saveDialog.ShowDialog() == DialogResult.OK:
            fileName = saveDialog.FileName
            if self.saveFile(fileName, text):
                self.document.fileName = filename

    def saveFile(self, fileName, text):
        try:
            writer = StreamWriter(fileName)          ❹
            writer.Write(text)
            writer.Close()
            return True
        except IOError, e:
            name = Path.GetFileName(fileName)
            MessageBox.Show(                         ❺
                'Could not write file "%s"\r\nThe error was:\r\n%s' %
                (name, e),
                "Error Saving File",
                MessageBoxButtons.OK,
                MessageBoxIcon.Error
            )
            return False
```

This listing shows the full code for the `SaveCommand`. The constructor creates and configures a `SaveFileDialog` ❶. It saves this, along with a reference to the document and the tab controller, as instance attributes for use whenever the command is launched.

The `execute` method is simple ❷. It fetches the text from the document, using the `getText` method, which also synchronizes the document by calling `updateDocument` on the tab controller.

It then checks to see if the document already has a filename set on it. If it does, then the code can jump straight to `saveFile`; otherwise, you need to prompt the user for a filename—which is done from `promptAndSave`. As well as checking whether a filename is set, it also checks that the directory specified still exists ❸; the file could have been on a USB key that has since been removed, meaning that the save will fail.

A lot of .NET methods allow calls with null values. You can pass `None` (the Python equivalent of null) to `Path.GetDirectoryName(fileName)`, and it returns `None` to you. When you then call `Directory.Exists(directory)`, where directory might now be `None`, `Exists` returns `False`. The code is a bit easier to read than making the conditional more complex.

`saveFile` attempts to save the file ❹, displaying the message box if anything goes wrong ❺. It returns `True` or `False` depending on whether or not it's successful. You can call it from `promptAndSave` (assuming the user doesn't cancel out of the file dialog box). If the Save File operation is successful, then you can set the filename on the document.

Because you've already written the `SaveCommand`, it ought to be easy to now build a `SaveAsCommand`.

4.3.5 The SaveAsCommand

The `SaveAsCommand` is going to share a lot of functionality with the `SaveCommand`. The main difference is that when the command is launched, it should always prompt the user to choose a filename. *But*, if the document does have a filename associated with it, the dialog box should start in the right directory and be prepopulated with this name. This is all done in listing 4.7.

Listing 4.7 `SaveAsCommand`, which needs to override dialog title and `execute` method

```
class SaveAsCommand(SaveCommand):          <-- Inherits from SaveCommand

   title = "Save Document As"              <-- Overrides title on SaveCommand

   def execute(self):
      fileName = self.document.fileName
      text = self.getText()
      if fileName is not None:             <-- Configures dialog initial paths
         name = Path.GetFileName(fileName)
         directory = Path.GetDirectoryName(fileName)
         self.saveDialog.FileName = name
         if Directory.Exists(directory):
            self.saveDialog.InitialDirectory = directory
      self.promptAndSave(text)             <-- Calls SaveAs
```

Thanks to splitting out a lot of the functionality of the Save command into separate methods, the Save As command is very little code! The change in functionality is all handled by overriding the title and the `execute` method. `execute` now sets initial state on the Save dialog box (if a filename already exists) and then always calls `promptAndSave`.

Let's pause and consider what you've achieved. The command pattern acts as a blueprint for creating new commands, so less thinking is required when you have to create, or wire up a new one! When you come to create a new command, you know what shape it should be and what interface you need to provide. You can also easily share functionality between similar commands.

Now that you've created the two commands Save and Save As, you need to hook them into the user interface so that the user has access to them.

4.4 Integrating commands with our running example

One way of providing access to the commands would be with a shortcut key. This is a useful feature (I'm a keyboard guy whenever possible), but users these days expect fancy GUI features. The standard way of providing access to commands like Save is through menus and toolbars. Fortunately, the .NET menu classes provide a way of setting shortcut keys on menu items, so you can have the best of both worlds.

So far, you've created a Model-View-Controller structure for our fledgling application. The view contains the GUI layout code. The model stores the underlying data, and the controllers (so far you only need one, for the tab control) mediates between the view and the model, implementing the logic required to keep them synchronized. Although the controllers hold a reference to the model, the model knows nothing about the view or the controllers and is completely independent.

User actions are represented by command classes, wrapping all the functionality into an object. Our commands have access to the tab controller and the model. They need to respond to the user interface and access the data model. In our case, you need to be able to ask the controller to update the document from the user interface.

For a small application like MultiDoc, this may feel like over-engineering. We've abstracted the application out into several classes, most of which are very small. But you've already seen, in the case of the Save As command, how this approach can be helpful; as you add new features, we can promise you'll appreciate this structure even more.

There's still one *vital* step missing: the commands aren't yet used. In this section, you'll extend the view by adding new ways for the user to interact with MultiDoc by launching the commands. Two obvious ways of doing this are through menus and a toolbar; we start with a menu.

4.4.1 Menu classes and lambda

Creating menus with Windows Forms is easy. You need to use different classes for the top-level menu strip and the individual menu item. .NET menus are controls like any of the other GUI elements used so far. You could put them anywhere in the MultiDoc form. The traditional place for menus, and the place where the user will expect them, is at the top of the form. You achieve this with the `Dock` property and `DockStyle.Top`.

The class we use for the top-level menu strip is `MenuStrip`. For the menu items, we use `ToolStripMenuItem`. As well as creating the menu hierarchy, this class allows you to specify shortcut keys to launch the individual items. The hierarchy we want to create, initially at least, is simple and looks like figure 4.7.

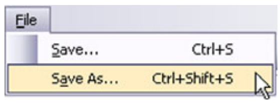


Figure 4.7 Menus created with `MenuStrip` and `ToolStripMenuItem`

Let's work from the bottom up, and start with the Save… and Save As... items. As with other controls, you set the displayed text of the item using the `Text` property.

```
>>> saveItem = ToolStripMenuItem()
>>> saveItem.Text = '&Save...'
```

Menu items have two types of keyboard shortcuts. The first type allows menus to be operated from the keyboard. Pressing Alt in a form displays the menu, which you can then navigate with the cursor keys. Items with an underlined letter can be launched with that key. You configure this shortcut by putting an ampersand (&) in front of a letter in the `Text` property.

The second key shortcut allows a menu item to be launched without the menu being open. The standard key combination for a save operation is Ctrl-S; you set this using the Windows Forms `Keys` enumeration. You combine the S key with the Ctrl key using a bitwise or the following:

```
>>> saveItem.ShortcutKeys = Keys.S | Keys.Control
```

The last thing you need to do for this menu item is to configure the action when it's launched, either by selection through the menu or by the keyboard shortcut. Here, we come back to our old friend the `Click` event, to which you need to add an event handler.

Obviously, we'd like our `SaveCommand` to be triggered, specifically the `execute` method. Unfortunately, there's a problem. Windows Forms event handlers receive two arguments when they're called, `sender` and `event`, but our `execute` method doesn't take any arguments. You can either rewrite `execute` to take two arguments, which it doesn't need, or you can wrap the call with another function which absorbs them.

Python has an easy way of creating functions in a single line: lambda functions.

NOTE Lambda functions are sometimes known as *anonymous functions* because, unlike the usual syntax for declaring functions, you don't need to give them a name. They can take arguments, but the body can only contain an expression and not statements. The expression is evaluated, and the result returned. Lambdas can be used anywhere that an expression is expected.

Lambda functions can be declared inline. The syntax is shown in figure 4.8.

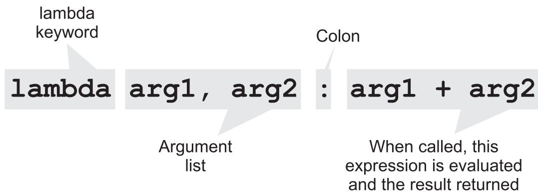


Figure 4.8 The syntax for the lambda function

You *could* hook up your commands by wrapping the call in a function like this:

```
>>> def handler(sender, event):
...     command.execute()
>>> item.Click += handler
```

Instead, use a lambda, which is identical (except shorter) to using a function:

```
>>> item.Click += lambda sender, event : command.execute()
```

Using a lambda here is appropriate because the function is only a convenience wrapper. You don't need to use it later, so it doesn't need a name.

You've now created and configured a menu item, but it still needs to be placed in its parent menu. For the Save item, the parent is the File menu. File is also a `ToolStripMenuItem`, with a `Text` property; its children (the submenu items) live in the `DropDownItems` collection.

```
>>> fileMenu = ToolStripMenuItem()
>>> fileMenu.Text = '&File'
>>> fileMenu.DropDownItems.Add(saveItem)
```

We're sure it won't surprise you to learn that the menu items will appear in the order that they're added. This isn't quite the end of the story; you still need to create a top-level menu and add the submenu to it. The top-level menu uses the `MenuStrip` class, and the submenus are added to its `Items` collection.

```
>>> menuStrip = MenuStrip()
>>> menuStrip.Items.Add(fileMenu)
```

Commonly used options are often exposed via a toolbar as well as menus. In .NET, you use the `ToolStrip` class.

4.4.2 .NET classes: ToolBar and images

Windows Forms contains a class called `ToolBar`. This is an abomination from .NET 1, and shouldn't be used under any circumstances! The right class for creating toolbars is the `ToolStrip`.

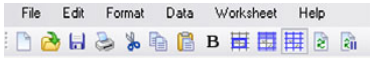


Figure 4.9 The Resolver One application toolbar

A toolbar has a series of icons that launch commands. Figure 4.9 shows the Resolver One toolbar.

A minor prerequisite for creating a toolbar is having some icons to use. My personal favorites are the free toolbar icons from glyFX.[8] After a great deal of scouring the internet, these are the best icons that I've found that you're free to use in applications. The usual condition for distributing these icons with your application is that you should use them as a compiled resource. You can do this by compiling the icons into your executable or by serializing the .NET image objects. We explore both of these techniques later in the book; but, for this example, glyFX has given us permission to use the icons without compiling them.[9]

Toolbars are another easy-to-use component. The toolbar itself is an instance of the `ToolStrip` class. The individual buttons are instances of `ToolStripButton`.

To associate an image with a button, you set the `Image` property with an instance of the `Bitmap` class, and set the `DisplayStyle` property to `ToolStripItemDisplayStyle.Image`. `Bitmap` comes from the `System.Drawing` assembly, and can be constructed from the path to the image file as a string.

> **The Bitmap class**
>
> The .NET `Bitmap` class is a useful class. You can use it for displaying images in applications, in conjunction with the Windows Forms `PictureBox` class.
>
> You can save images, in any of the multitude of formats that .NET supports, using the `Save` method. You can pass in a filename and a value from the `ImageFormat` enumeration to specify the format.
>
> You can examine the contents of the image with the `GetPixel` method, which takes X and Y coordinates and returns a color.
>
> Conversely, you can create a new bitmap by constructing with two integers representing the X and Y dimensions of the image. You can then create the image programmatically by passing in coordinates and a color to the `SetPixel` method.

The icon for Save is a 16x16 pixel icon called save_16.ico, which is stored in a folder called icons. The path to the icon, relative to our application file, is icons/save_16.ico.

---

8 See http://www.glyfx.com/products/free.html.

9 They're included in the downloads available from the *IronPython in Action* website at http://www.ironpythoninaction.com.

At runtime, you have no idea what the current directory is; you can't assume that it will be the application directory.

If you were running with a custom executable, you could extract the directory from the full path to the executable.

```
>>> from System.Windows.Forms import Application
>>> from System.IO import Path
>>> executablePath = Application.ExecutablePath
>>> directory = Path.GetDirectoryName(executablePath)
```

If your script is being run by ipy.exe, then this approach will return the directory containing ipy.exe. When run in this way, a magic name called `__file__` is available with the full path of the file currently being executed. In that case, you can replace `executablePath` with `__file__`.

The following code segment should work in either case:

```
executablePath = __file__
if executablePath is None:
   executablePath = Application.ExecutablePath
directory = Path.GetDirectoryName(executablePath)
```

Now that you can reliably work out the path to the icon, you can create the toolbar button.

```
>>> from System.Drawing import Bitmap, Color
>>> from System.Windows.Forms import ToolStripButton, ToolStripItemDisplayStyle
>>> iconPath = Path.Combine(directory, r'icons\icon.bmp')
>>> image = Bitmap(iconPath)
>>> button = ToolStripButton()
>>> button.DisplayStyle = ToolStripItemDisplayStyle.Image
>>> button.Image = image
```

Toolbar icons are usually intended to be displayed with one color nonvisible; this is a cheap way of gaining a transparency layer. The convention is that `Color.Magenta` is used as the transparent color. This is an odd convention, but must be because magenta is such a ghastly[10] color that it would be rare to want to see it…

```
>>> button.TransparentColor = Color.Magenta
```

As well as setting the image on the toolbar button, you can set the text that will be displayed when the mouse hovers over the button, known as the tooltip text:

```
>>> button.ToolTipText = 'Save'
```

You set the command on the button in exactly the same way as you did for the menu.

```
>>> button.Click += lambda sender, event: command.execute()
```

The toolbar buttons are added to the `Items` collection of the `ToolStrip`. Again we want the toolbar to appear at the top of the form, just below the menu strip. To do

[10] Which is perhaps unfair for a color that has such a noble history. It was one of the few colors my first computer could display. The name magenta also has a history both more noble and ghastly than you might realize. It was named after the Battle of Magenta in Italy in 1859.

this, you still use `DockStyle.Top`, but you have to add the toolbar to the form *after* the menu. The toolbar will then be docked to the bottom of the menu.

```
>>> toolbar = ToolStrip()
>>> toolbar.Items.Add(button)
>>> toolbar.Dock = DockStyle.Top
```

By default, the toolbar comes with a gripper, which can be used to move it around. But, also by default, you can't move the toolbar around, so it makes sense to hide it. You do this with another wonderful Windows Forms enumeration: the `ToolStripGripStyle`.

```
>>> toolbar.GripStyle = ToolStripGripStyle.Hidden
```

Now you know enough to use both toolbars and menus in MultiDoc.

4.4.3 Bringing the GUI to life

You've now written your commands, and learned all you need to know to wire them into the MultiDoc user interface. You need to make some changes in the presentation layer and our `MainForm` class, as well as some additions to the imports to support the new code.

Listing 4.8 shows the expanded import code for the new and improved MultiDoc (hey, we're getting there!). It includes the code that determines the path to the project directory, so that you can load the icon.

Providing a menu and toolbar means using lots of extra names from the Windows Forms namespace, a lot of which are enumerations for configuration. You could replace the long list of names with `from System.Windows.Forms import *`. This wouldn't show you explicitly which names you're using (and would pollute the namespace with a lot more names that you're *not* using); personally, I think it's bad practice. Jim Hugunin disagrees with me, so you'll have to decide whom you trust more![11]

Listing 4.8 Full import code for expanded `MainForm`

```
import clr
clr.AddReference('System.Drawing')
clr.AddReference('System.Windows.Forms')

from System.Drawing import Bitmap, Color, Size
from System.IO import Directory, Path, StreamWriter
from System.Windows.Forms import (
   Application, DialogResult, DockStyle, Form,
   Keys, MenuStrip, MessageBox, MessageBoxButtons,
   MessageBoxDefaultButton, MessageBoxIcon,
   ScrollBars, SaveFileDialog, TabAlignment,
   TabControl, TabPage, TextBox,
   ToolStripItemDisplayStyle, ToolStrip,
```

[11] In fact, Jim is virtually alone amongst experienced Pythonistas in advocating `from module import *`, but he does have an interesting point. He argues that, instead of polluting the current namespace, it should add the namespace of the imported module to those searched when names are looked up, behavior which matches the semantics of other languages such as VB.

```
    ToolStripButton, ToolStripGripStyle,
    ToolStripMenuItem
)

executablePath = __file__        <-- Determines project directory
if executablePath is None:
    executablePath = Application.ExecutablePath
executableDirectory = Path.GetDirectoryName(executablePath)
```

You've accumulated all the knowledge you need to integrate the commands into `MainForm`, a process that will change it almost out of recognition. Previously, it was a simple class, only twenty-four lines of code initializing a couple of components, barely deserving of its own class. Here, the view comes into its own, and the new additions to `MainForm` swell it substantially. We've split the changes into several segments to make them easier to digest. Once you've pieced all these changes together, not forgetting the commands and model classes, running MultiDoc should result in something that looks remarkably like figure 4.10. Clicking the toolbar icon, or using the menu options, should bring up the Save File dialog box.



Figure 4.10 The MultiDoc Editor with Save and Save As commands, using a menu and a toolbar

Listing 4.9 shows the new `MainForm` constructor. You need to add the calls to new methods to initialize the commands, menus, and toolbar. Both the toolbar and top-level menu are positioned using `DockStyle.Top`. To get the desired layout (menu at the top of the form, followed by the toolbar), the toolbar must be initialized *before* the menu.

The `initializeCommands` method is trivially simple, so we've included it here. We'll go through the next two methods individually.

Listing 4.9 New `MainForm` constructor and `initializeCommands`

```
class MainForm(Form):
    def __init__(self):
        Form.__init__(self)
        self.Text = 'MultiDoc Editor'
        self.MinimumSize = Size(150, 150)

        tab = self.tabControl = TabControl()
        self.tabControl.Dock = DockStyle.Fill
        self.tabControl.Alignment = TabAlignment.Bottom
        self.Controls.Add(self.tabControl)

        doc = self.document = Document()
        self.tabController = TabController(tab, doc)

        self.initializeCommands()
        self.initializeToolbar()
        self.initializeMenus()
```

```
def initializeCommands(self):          <-- Sets up commands
    tabC = self.tabController
    doc = self.document
    self.saveCommand = SaveCommand(doc, tabC)
    self.saveAsCommand = SaveAsCommand(doc, tabC)
```

The code for creating new menu items and new toolbar buttons is simple, but a bit verbose. It would be tedious to have to type all the code for each new menu item and toolbar icon. Both `initializeToolbar` and `initializeMenus` follow the same pattern; a top-level method creates the main control, which calls into a submethod for creating each of the items. The two methods for creating the toolbar are shown in listing 4.10.

Here the submethod is called `addToolbarItem`. The top-level control is the `ToolStrip`, which is configured and added to the form's collection of controls. The toolbar icon needs some text to display as the tooltip, a function to call for the click handler, and a path to the icon file. These three things are the arguments that `addToolbarItem` takes.

Listing 4.10 Methods to create main toolbar and Save button

```
def initializeToolbar(self):
    self.iconPath = Path.Combine(executableDirectory, 'icons')
    self.toolBar = ToolStrip()
    self.toolBar.Dock = DockStyle.Top
    self.toolBar.GripStyle = ToolStripGripStyle.Hidden

    self.addToolbarItem('Save',
                  lambda sender, event: self.saveCommand.execute(),
                     'save_16.ico')                          <-- Adds Save button
    self.Controls.Add(self.toolBar)                          <-- Adds toolbar to form

def addToolbarItem(self, name, clickHandler, iconFile):
    button = ToolStripButton()
    button.Image = Bitmap(Path.Combine(self.iconPath, iconFile))
    button.ImageTransparentColor = Color.Magenta
    button.ToolTipText = name
    button.DisplayStyle = ToolStripItemDisplayStyle.Image
    button.Click += clickHandler

    self.toolBar.Items.Add(button)
```

Creating the menu is similar; the method `createMenuItem` is responsible for creating the individual menu items. There's a slight complication—the class used to represent the individual items is *also* used to represent the container menus like File. Sometimes `ToolStripMenuItem` needs to be configured with a click handler, and sometimes without. To get around this, the `handler` and `keys` (for the shortcut key) arguments to `createMenuItem` are optional. Listing 4.11 shows the full code for creating the menus.

Listing 4.11 Methods to create menu strip and submenus

```
def initializeMenus(self):
    menuStrip = MenuStrip()          <-- Main menu strip
```

```
        menuStrip.Dock = DockStyle.Top

        fileMenu = self.createMenuItem('&File')      <- Creates File menu

        saveKeys = Keys.Control | Keys.S
        saveMenuItem  = self.createMenuItem(
           '&Save...',
           handler  = lambda sender, event: self.saveCommand.execute(),
           keys=saveKeys
        )                           <- Creates Save menu item

        saveAsKeys = Keys.Control | Keys.Shift | Keys.S
        saveAsMenuItem  = self.createMenuItem(
           'S&ave As...',
           lambda sender, event: self.saveAsCommand.execute(),
           keys=saveAsKeys
        )                              <- Creates Save As menu item

        fileMenu.DropDownItems.Add(saveMenuItem)     <- Adds items to File
        fileMenu.DropDownItems.Add(saveAsMenuItem)

        menuStrip.Items.Add(fileMenu)
        self.Controls.Add(menuStrip)

    def createMenuItem(self, text, handler=None, keys=None):
        menuItem = ToolStripMenuItem()
        menuItem.Text = text

        if keys:
           menuItem.ShortcutKeys = keys
        if handler:
           menuItem.Click += handler
        return menuItem
```

As you can see, even with adding just this basic functionality, `MainForm` has grown quite a bit. It's now recognizable as an application, and it actually does something! To get this far, you've used .NET classes from several different namespaces, and you have the structure in place to make adding more features easy.

4.5 Summary

In this chapter, you've created the beginnings of the MultiDoc application. Along the way, you've learned new things about Python, such as the use of lambda functions, and used .NET classes such as `Path` and `StreamWriter`. Through the use of design patterns, you've provided infrastructure for adding menus and toolbar items, keeping the data model and the view in sync, and for saving the text from a single page.

MultiDoc still doesn't fulfill the specification, though. You still need to add the following features:

- The ability to add and remove tab pages
- Saving of multipage documents
- Loading of documents

You could add many more features, just to provide what people think of as essential requirements in such a basic program. Writing an application is a big task. Even so,

thanks to the succinctness of Python, MultiDoc currently stands at less than 250 lines of code including blank lines.

To add extra tabs without *losing* functionality, you'll need to be able to save multipage documents. Our basic text format won't quite cut it, so we start the next chapter with a writer that can handle multiple pages. On top of this, you'll add a document loader, which will require the use of another design pattern, and a command to open documents.

First-class functions in action with XML

This chapter covers

- First-class and higher order functions
- Python decorators
- Reading and writing XML
- An XML file format for MultiDoc

In the previous chapter, we started creating a multipage document editor called MultiDoc. The aim is to create a desktop application that can load and save multipage documents stored in a single file. In this chapter, we show the implementation of a core part of this specification: the ability to save and load multipage documents stored as XML. You'll add new load and save commands to MultiDoc, using classes in the `System.Xml` assembly. In the process, we'll explore an extensible approach to reading XML with IronPython.

One of the big differences between Python and traditional .NET languages is its support for first-class functions. We start this chapter by looking at what first-class functions are and how they can help you write shorter and more beautiful code.

Next, we turn our attention to working with the .NET XML classes and putting what you've learned about functions to work.

First, on with the first-class functions.

5.1 First-class functions

In a programming language, functions are first class if they're just another kind of object—you can create them at runtime and pass them around your code, including using them as arguments to functions or return values from functions.

First-class functions are a core part of the functional-programming style of programming. Functional programming breaks problems into a set of functions. Preferably, these functions should only have inputs and outputs, neither storing internal state nor having side effects.

To programmers used to working with object-oriented languages, this seems an odd set of constraints—almost a step backwards. But there are some advantages to this style. Along with encouraging modularity, elegance and expressiveness are the biggest advantages. It's also theoretically possible to construct formal proofs of purely functional programs—which has attracted a lot of interest in academic circles.

One aspect of first-class functions that you met in the Python tutorial is inner functions. These are functions defined inside the body of another function or method. If you have code repeated inside a function, it's often tidier to factor this out into another function. If the code uses several local variables, then it may require too many arguments if turned into a separate method. In this case, it can make sense for it to be an inner function that has access to any of the variables in the scope in which it's defined.

There's another class of functions commonly used in functional programming: *higher order functions*.

5.1.1 Higher order functions

Functions that take functions as arguments or return functions are called *higher order functions*. Functions that work with functions can be extremely useful. They allow you to separate out parts of a program more easily; for example, a higher order function might provide a traversal of a data structure, where the function you pass in decides what to do for each item.

This technique is highly reminiscent of the strategy pattern from *Design Patterns: Elements of Reusable Object-Oriented Software*:[1]

> *Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.*

You can implement this pattern very simply with higher order functions. Imagine you're writing a banking application that handles many different transactions. You

[1] The strategy pattern is the last pattern described in this 1995 book.

need to apply these transactions to accounts, rolling back the transactions in the event of an error (such as insufficient funds in one of the accounts). Transactions of many different types can be created, but the code that consumes the transactions should be able to apply them without knowing anything about how they're implemented.

A function to create a transaction may look something like this:

```
def get_transfer(account1, account2, amount):
   def do_transfer():
      account1.withdraw(amount)
      account2.deposit(amount)

   return do_transfer
```

The `do_transfer` function is for transactions that transfer money from one account to another. `do_transfer` closes over the parameters to `get_transfer` and encapsulates them. The parameters will be used when `do_transfer` is applied.

Functions and the __call__ method

Functions are only one example of callable objects in Python. You can create your own callable objects, which behave like functions, by implementing the `__call__` method.

```
class Callable(object):
   def __call__(self, x):
      print x
```

Instances of the `Callable` class are callable like functions.

```
>>> instance = Callable()
>>> instance('hello')
hello
```

The transaction is applied by `apply_transaction`.

```
def apply_transaction(transaction, connection):
   connection.begin_transaction()
   try:
      transaction()
   except TransactionError:
      connection.abort_transaction()
      return False
   else:
      connection.commit_transaction()
      return True
```

`apply_transaction` can apply any transaction. It handles errors, aborting the transaction in case of error. It returns `True` or `False` depending on whether the transaction succeeds or not. The advantage of this kind of structure is that you can change how the transactions are applied in a single place, and individual transactions can be modified without affecting how they're used.

Another place where higher order functions commonly turn up in Python is in the form of decorators.

5.1.2 Python decorators

It turns out that writing a function that takes a function, and then wraps it, is a common idiom in Python. Python has syntax to make this easier, syntactic sugar called *decorators*.

First, we show you an example that doesn't use the decorator syntax. We wrap a function so that it prints whenever it is called. In practice, you might put logging or timing code inside the wrapper function.

```
>>> def somefunction():
...     return "From somefunction"
...
>>> def wrapper(function):
...     def inner():                    <-- Inner function
...         print "From the wrapper"
...         return function()           <-- Calls wrapped function
...     return inner
...
>>> wrapped = wrapper(somefunction)       <-- Wraps somefunction
>>> print wrapped()
From the wrapper
From somefunction
```

The decorator syntax allows a nicer way of expressing the line `wrapped = wrapper (somefunction)`. You'll especially appreciate it if you want to decorate all the methods in a class. Decorators use the @ symbol, along with the name of the decorator function, above the function definition.

```
>>> @wrapper                  <-- Marks somefunction as decorated
>>> def somefunction():
...     print "I've been called"
...
>>> somefunction()
From the wrapper
I've been called
```

When you decorate a function, the function name is automatically rebound to the *wrapped* function rather than the original one. So is this actually useful? Let's look at how you can use this to automate a repetitive task like checking method arguments for null values.

5.1.3 A null-argument-checking decorator

Even with static typing, method parameters can still be `None` (or null, using .NET speak) in .NET languages such as C#. If your code requires that the argument is valid and can't be null, then it's common to have code like the following:

```
void Present(Region present, Region selection)
{
  if (present == null)
  { throw new ArgumentNullException("present"); }
  if (selection == null)
  { throw new ArgumentNullException("selection"); }

  // Actual code that we care about
}
```

In Python, you can write a decorator that checks function arguments and raises a `TypeError` if any of them are `None`. Any methods that you want to check in this way can then be decorated.

To do this, you need a decorator that returns a wrapped function. The wrapper should check all the arguments before calling the function or method that's wrapping. The wrapper function will need to call the wrappee with the *same* arguments it's called with, and return the value that the wrapped function returns. This should do the trick:

```
def checkarguments(function):
   def decorated(*args):
      if None in args:
         raise TypeError("Invalid Argument")
      return function(*args)
   return decorated
```

Any arguments a wrapped function is called with are collected as a tuple (`args`). If any of these arguments are `None` (if `None` is in `args`), then a `TypeError` is raised. You use it like this:

```
class MyClass(object):

   @checkarguments
   def method(self, arg1, arg2):
      return arg1 + arg2
>>> instance = MyClass()
>>> instance.method(1, 2)
3
>>> instance.method(2, None)
Traceback (most recent call last):
TypeError: Invalid Argument
```

You should pass the parameter name when you raise the exception. You could do this by using introspection on the function object, but that's another subject altogether.

OK, we've had some fun with functions in Python. Now it's time to put what you've learned to practical use in working with XML.

5.2 Representing documents with XML

XML is a text-based format that uses tags to structure data. XML is certainly no silver bullet when it comes to persisting data; it's generally verbose and, for complex data structures, inefficient. On the other hand, XML *is* a text-based format. If something goes wrong with your program, it's easier to *see* what's going wrong than with a binary format.

An XML document will look something like the following segment:

```
<?xml version="1.0" encoding="utf-8"?>
<rootelement>
   <childelement1 attribute="An attribute value.">An element value, with parsed
     character data.</childelement1>
   <childelement2><![CDATA [An element value, with unparsed character data.
     ]]></childelement2>
</rootelement>
```

The `encoding` declaration at the start is optional, but useful if the document is in an encoding other than ASCII or UTF-8. XML is a hierarchical format, so it must contain a root element, which contains the rest of the document. The root element can contain child elements, which themselves can contain child elements nested to whatever degree of horrific complexity you desire.

XML elements are simple, but wrap together several concepts. Figure 5.1 shows all the components of an element.

The text content contained between an element start tag and the end tag is restricted. Angle brackets in the text could be confused for the start of a new tag, unless they're escaped. The XML spec stipulates five characters that should be escaped with entity references. Table 5.1 shows the characters that need to be escaped and their corresponding entity references.

Table 5.1 XML escaped characters and their corresponding entity references

Character	Entity reference	Name
<	`<`	less than
>	`>`	greater than
&	`&`	ampersand
'	`'`	apostrophe (quoting only mandatory in attribute values)
"	`"`	quotation mark (quoting only mandatory in attribute values)

Good XML writing and parsing libraries will handle the escaping and un-escaping for you; but, if your text contains a lot of these characters, it can make the resulting XML much less human-readable. Plain text content, which needs escaping and un-escaping, is called parsed data. An alternative is to include the text content in unparsed character data blocks, like the one from the previous XML document:

```
<![CDATA[An element value, with unparsed character data. ]]>
```

But there's a disadvantage to CDATA blocks. They *must not* contain `]]>` inside the text, and there's no way of escaping them if they do occur. If you need to include arbitrary text—sourced from a user, for example—then CDATA blocks may not be suitable.

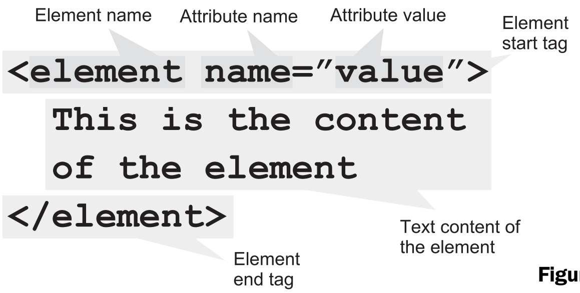


Figure 5.1 An XML element in all its glory

Back to the subject at hand: representing documents with XML. `document` will make an excellent root level element. You also need an element to represent pages. Each page has a title and contents. It seems logical that the title should be an attribute of the `page` start tag, and that the page contents should be the text content between the `start` and `end` elements. This will leave you with documents that look like the following segment:

```
<?xml version="1.0" encoding="utf-8"?>
<document>
   <page title="The page title">This is the contents of the first page.</page>
    <page title="Page Two">This is another page, as beautiful as the first.</
      page>
</document>
```

Now that the structure for representing documents is decided, you need to write the code to turn the model objects into XML.

5.2.1 The .NET XmlWriter

The .NET XML support is enshrined in the `System.Xml` assembly. The `System.Xml` namespace has classes for reading and writing XML. There are further namespaces for working with XML schema, XSLT transformations, and XPath queries. Table 5.2 shows the major XML namespaces in .NET 2.0.

Table 5.2 The .NET XML namespaces

Namespace	Purpose
`System.Xml`	Provides support for standards-based XML support. Includes the `XmlReader` and `XmlWriter` classes.
`System.Xml.Schema`	Support for XML Schemas in the form of schema definition language (XSD) schemas.
`System.Xml.Serialization`	Classes to serialize objects in XML form.
`System.Xml.XPath`	Classes to work with the XQuery 1.0 and XPath 2.0 Data Model.
`System.Xml.Xsl`	Support for Extensible Stylesheet Language Transformation (XSLT) transforms.

The basic classes for reading and writing XML documents are the `XmlReader` and `XmlWriter`. Because you'll be modifying the `SaveCommand`, we start with the `XmlWriter`.

`XmlWriter` is designed for creating conformant documents. The documents it creates will be valid XML, capable of being read by any standards-based reader. Along with the `XmlWriter` class, you use `XmlWriterSettings`. This is a class used for configuring an `XmlWriter` instance; you set attributes on the `XmlWriterSettings` instance to configure how the XML is written out.

Table 5.3 shows the different settings (properties) on `XmlWriterSettings`. The defaults are largely sensible, but we do like to change a couple. We like the XML tags

Table 5.3 The properties of `XmlWriterSettings` and the default values

Property	Initial value
`CheckCharacters`	`True`
`CloseOutput`	`False`
`ConformanceLevel`	`Document`
`Encoding`	`Encoding.UTF8` (`Encoding` lives in the `System.Text` namespace, and is a useful class.)
`Indent`	`False`
`IndentChars`	Two spaces
`NewLineChars`	`\r\n` (carriage return, new line)
`NewLineHandling`	`Replace`
`NewLineOnAttributes`	`False`
`OmitXmlDeclaration`	`False`

to be indented with each level of nesting. This gives you a visual indication of the structure of the document (and we all know that indentation to indicate structure is a brilliant idea). The following segment creates an `XmlWriterSettings` instance, and sets the two properties required for indentation with four spaces. Because you haven't yet used the `System.Xml` assembly, you first need to add a reference to it.

```
>>> import clr
>>> clr.AddReference('System.Xml')
>>> from System.Xml import XmlWriter, XmlWriterSettings
>>> settings = XmlWriterSettings()
>>> settings.Indent = True
>>> settings.IndentChars = '    ' # four spaces
```

You don't instantiate a new `XmlWriter` instance directly; instead, you call the static method `Create`, which returns a new instance of the correct type of writer for the settings passed in. There are various call signatures for `Create`; for example, you could pass in a filename and your settings object, and the writer would create the file for you. If you don't want the writer to be responsible for creating the file, you can pass in an opened `FileStream` instead. In the segments that follow, you'll pass in a `StringBuilder` instance. `StringBuilder` is in the `System.Text` namespace, and is a mutable string type—it allows strings to be built up incrementally.

An odd side effect of passing a `StringBuilder` to the `XmlWriter` is that it will refuse to write any encoding in the XML declaration other than UTF-16.[2] Because

[2] Because the writer is writing into a string, which is still Unicode and has no encoding yet, it's no wonder that it gets confused. The logic is *possibly* that the Windows internal UCS2 Unicode representation is most like UTF-16. Still, ignoring the explicit encoding on the `XmlWriterSettings` is a dubious practice.

you're likely to be happy with the default encoding of UTF-8, you set `OmitXmlDeclaration` to `True`.

```
>>> settings.OmitXmlDeclaration = True
>>> from System.Text import StringBuilder
>>> document = StringBuilder()
>>> writer = XmlWriter.Create(document, settings)
>>> writer.WriteStartDocument()
>>> writer.WriteStartElement("document")
>>> writer.WriteStartElement("page")
>>> writer.WriteAttributeString("title", "A page title")
>>> writer.WriteString("This is a page contents")
>>> writer.WriteEndElement()
>>> writer.WriteEndElement()
>>> writer.WriteEndDocument()
>>> writer.Flush()
>>> writer.Close()
>>> print document.ToString()
  <document>
   <page title="A page title">This is a page contents</page>
  </document>
```

This is great because (by happy coincidence) it's exactly the kind of structure that we want for our documents. Having to remember to close all the elements in the right order is a nuisance, though. This is what happens if you get it wrong:

```
>>> document = StringBuilder()
>>> writer = XmlWriter.Create(document, settings)
>>> writer.WriteEndElement()
Traceback (most recent call last):
  File System.Xml, line unknown, in WriteEndElement
  File System.Xml, line unknown, in AdvanceState
  File System.Xml, line unknown, in ThrowInvalidStateTransition
SystemError: Token EndElement in state Start would result in an invalid XML document.
Make sure that the ConformanceLevel setting is set to ConformanceLevel.Fragment or
ConformanceLevel.Auto if you want to write an XML fragment.
```

Oops. A sensible way to avoid this is to make sure that your XML document structure and your program structure are as similar as possible. Ideally, a top-level method should create (and close) the root node, calling down to submethods to write out its child elements. Each method should only be responsible for creating and closing a single element, again calling submethods for their child elements. This way you know that, as long as the code runs to completion, every element will be closed and the result will be valid XML. A nice side effect is that this is also a good way to write modular and readable code—which is important because you're writing Python. In the next section, we apply this strategy to MultiDoc.

5.2.2 A DocumentWriter Class

Writing out XML from a MultiDoc document is an important enough job that we can encapsulate it in its own class. This keeps the logic separate from the other machinations of the save commands, and easier to understand.

NOTE By the end of chapter 4, we were still keeping the MultiDoc project as a single file, which was getting large and unwieldy. To make it easier to work with and provide a better application structure, you can break MultiDoc down into several modules containing the core classes. You can see this structure if you download the source code that accompanies this book. All the code from here on contains the appropriate imports to use our classes from the correct modules.

Listing 5.1 shows the `DocumentWriter` for MultiDoc; it uses `XmlWriter` and is instantiated with a filename. To write out the document, you must call `write`, passing in an instance of a MultiDoc document. The `write` method is responsible for creating the `XmlWriter` instance and opening and closing the root element of the XML. It calls down to `writePage` for each page in the document, creating the `page` element with `title` attribute.

Listing 5.1 A `DocumentWriter` class that writes out MultiDoc documents as XML

```
from System.Xml import XmlWriter, XmlWriterSettings

class DocumentWriter(object):

    def __init__(self, fileName):
        self.fileName = fileName

    def write(self, document):
        settings = XmlWriterSettings()                          # Configures XML settings
        settings.Indent = True
        settings.IndentChars = '    '
        settings.OmitXmlDeclaration = True
        self.writer = XmlWriter.Create(self.fileName, settings)  # Creates writer object

        self.writer.WriteStartDocument()
        self.writer.WriteStartElement("document")

        for page in document:                                   # Writes each page in document
            self.writePage(page)

        self.writer.WriteEndElement()
        self.writer.WriteEndDocument()
        self.writer.Close()

    def writePage(self, page):
        self.writer.WriteStartElement("page")
        self.writer.WriteAttributeString("title", page.title)
        self.writer.WriteString(page.text)
        self.writer.WriteEndElement()
```

This class needs to be saved as the file documentwriter.py. To plug this into MultiDoc, you need to modify the commands to use it. First, `DocumentWriter` needs to be imported inside the `savecommands` module.

```
from documentwriter import DocumentWriter
```

The `SaveAsCommand` inherits from `SaveCommand`, and the file writing is done in the `saveFile` method. You can get most of the way toward the changes you want by modifying `saveFile` as follows:

```
def saveFile(self, fileName, document):
      try:
         writer = DocumentWriter(fileName)
         writer.write(document)
         return True
      except IOError, e:
         ...
```

Previously the `execute` and `promptAndSave` methods of the commands only needed to pass some text to `saveFile`; now they need to pass the document instead. The document still needs to be updated before saving, so `getText` becomes `getUpdatedDocument`.

```
def getUpdatedDocument(self):
   self.tabController.updateDocument()
   return self.document
```

The `execute` method of both `SaveCommand` and `SaveAsCommand` must be modified to call `getUpdatedDocument`, and to pass the document through to `saveFile` and `saveAndPrompt`. These changes are simple; and, rather than using space here we leave you to figure them out. If you want to see the changes, they're in the 5.3 folder of the sources that go with this book.

The `DocumentWriter` you've created follows the structure we suggested earlier, with a top-level method that writes out the top-level node (`write`). This calls down to `writePage` to write out the child page nodes. To do this, the `XmlWriter` has to be stored as state on the `DocumentWriter`, as the `self.writer` instance variable. Because you only have one root node, the structure is simple. You could avoid having to store state by refactoring to use an inner function.

5.2.3 An alternative with an inner function

The `writePage` method is simple. You can refactor this into an inner function that takes a `page` as its argument. It needs access to the writer, which can be a local variable in its enclosing scope, which is the body of the `write` method.

The refactored `write` method looks like listing 5.2.

Listing 5.2 Implementation of `DocumentWriter.write` using an inner function

```
def write(self, document):
   settings = XmlWriterSettings()
   settings.Indent = True
   settings.IndentChars = '    '
   settings.OmitXmlDeclaration = True
   writer = XmlWriter.Create(self.fileName, settings)

   writer.WriteStartDocument()
   writer.WriteStartElement("document")

   def WritePage(page):
      writer.WriteStartElement("page")
      writer.WriteAttributeString("title", page.title)
      writer.WriteString(page.text)
      writer.WriteEndElement()
```

```
for page in document:
    WritePage(page)

writer.WriteEndElement()
writer.WriteEndDocument()
writer.Flush()
writer.Close()
```

This version of `write` is still an acceptable length, and the number of occurrences of `self` have been dramatically reduced, making the code more readable!

TIP There's a performance implication when defining inner functions. The `def` statement is executed every time the method containing it is executed. It isn't a high cost, but may become significant if it's in a performance-sensitive part of your code. Inner functions close over the variables in their containing scope. They're most useful when you need a new closure for each execution.

Now that we've created a way of writing MultiDoc documents in XML formats, we ought to provide a way of reading them back in again.

5.3 Reading XML

The counterpart to `XmlWriter` is `XmlReader`. Although logically it's the inverse of `XmlWriter`, `XmlReader` is slightly more complex; it has almost twice as many public properties and methods. A lot of these are to allow you to read typed data from an XML file.

5.3.1 XMLReader

`XmlReader` is usually instantiated in the same way as `XmlWriter`—through the static `Create` method. There's a plethora of different overloads for creating new instances; you can supply a stream, `TextReader`, or a resource locator as a string. You can also optionally pass in `XmlReaderSettings` and an `XmlParserContext`, and just about any combination of these items.

NOTE There are two common approaches to parsing XML. The first, perhaps more intuitive, is to read the whole document and access the Document Object Model in memory. This is known as DOM parsing, and it turns out to be very cumbersome in practice. This chapter uses event-driven parsing, which fires events as elements of the document are read in.

The resource locator doesn't need to be a filename; it can also be a URI[3] so that the XML document can be fetched from the internet. The default `XmlResolver` used by `XmlReader` is an `XmlUrlResolver`, which supports URIs that use the http:// and file:// protocols. You can supply authentication credentials, or use a different resolver by setting it on the instance of `XmlReaderSettings` that you use to `Create` your `XmlReader`.

[3] Uniform Resource Indicator—a term often used interchangeably with URL (Uniform Resource Locator), but supposedly more general.

The most convenient way, especially for this use case, is to supply a filename and an `XmlReaderSettings` instance. Table 5.4 shows the configurable properties on `XmlReaderSettings`, along with their default values. If you were reading only a fragment of XML, you might want to set `ConformanceLevel` to `ConformanceLevel.Fragment`. If the XML could have processing instructions that you don't want to handle, then you can set `IgnoreProcessingInstructions` to `True`.

Table 5.4 The properties of `XmlReaderSettings` and the default values

Property	Initial value
`CheckCharacters`	`True`
`ConformanceLevel`	`ConformanceLevel.Document`
`IgnoreComments`	`False`
`IgnoreProcessingInstructions`	`False`
`IgnoreWhitespace`	`False`
`LineNumberOffset`	`0`
`LinePositionOffset`	`0`
`NameTable`	`None`
`ProhibitDtd`	`True`
`Schemas`	An empty `XmlSchemaSet` object
`ValidationFlags`	`ProcessIdentityConstraints` enabled
`ValidationType`	`ValidationType.None`
`XmlResolver`	A new `XmlUrlResolver` object

Curiously, the default is for the `XmlReader` to *not* ignore insignificant whitespace (for example, whitespace that indents elements rather than being part of an attribute value or element content). Because you probably won't want to handle insignificant whitespace, the following code segment shows the pattern you'll be using for creating and configuring an `XmlReader`:

```
from System.Xml import XmlReader, XmlReaderSettings
settings = XmlReaderSettings()
settings.IgnoreWhitespace = True
reader = XmlReader.Create(filename, settings)
```

Once you've opened the XML file with `XmlReader`, the most straightforward way to use your instance is to repeatedly call the `Read` method. This consumes the document, one node at a time, exposing information about each node.

On completion, you should call `reader.Close()`, which frees the file. Forgetting to do this will cause the file to be held open until the reader is garbage-collected.

If you take the straightforward approach to reading MultiDoc files, the code would be simple, but also tedious and trivial. To make this section more interesting (and possibly even useful), we look at a more general approach to reading XML documents.

5.3.2 An IronPython XmlDocumentReader

A useful pattern for implementing a `Read` loop is to establish handlers for the *types* of node that you expect. `Read` can be called in a simple loop, delegating to the handlers for each node you encounter.

Our documents have elements, with or without attributes and with or without contents. This doesn't use the whole XML spec, not by a long stretch of the imagination. XML has *many* additional aspects to it, such as CDATA, namespaces, and processing instructions. You could write more handlers for these components we haven't yet dealt with, but a significant proportion of XML documents are made up of nothing more than elements, attributes, and text.

The following code is a general `XmlDocumentReader` class. It's adapted from a similar class used in Resolver One; many thanks to the kind Resolver Systems folk for letting us use and abuse the code. We've mangled it quite a bit from the original, so any bugs are entirely of our own devising. As well as being a general (and easily extensible) XML reading class, it's another example of using Python functions as first-class objects. The node handlers are packaged as a dictionary of node types mapping to functions (or methods) that know how to process them.

Listing 5.3 shows the imports and the constructor for `XmlDocumentReader`. It's instantiated with the node handling functions. These will be explained when we get to using them, and we'll put together a concrete example for reading MultiDoc documents.

Listing 5.3 Importing code and constructor for `XmlDocumentReader`

```
import clr
clr.AddReference('System.Xml')

from System.Xml import (
   XmlException, XmlNodeType,
   XmlReader, XmlReaderSettings
)

MISSING_HANDLERS = (None, None, None)

class XmlDocumentReader(object):
   def __init__(self, elementHandlers):
      self._elementHandlers = elementHandlers
```

Deals with unrecognized elements (MISSING_HANDLERS)

Dictionary mapping handlers to element types (self._elementHandlers)

Listing 5.4 is the `read` method of `XmlDocumentReader`. The element handlers passed into the constructor will be called when the reader encounters different elements in the document. This happens within the `onStartElement` method, which is called whenever an element start tag is encountered. Because you're just handling elements, attributes, and contents, you need only three general node handlers: element start tags, element end tags, and the text contents of elements. Element attributes will be dealt with *inside* the element start tag node handler (listing 5.4).

Listing 5.4 `XmlDocumentReader` `read` method

```
def read(self, filename):
   settings = XmlReaderSettings()
   settings.IgnoreWhitespace = True
   reader = XmlReader.Create(filename, settings)

   self._currentElement = None

   nodeTypeHandlers = {
      XmlNodeType.Element    : self.onStartElement,
      XmlNodeType.EndElement : self.onEndElement,
      XmlNodeType.Text       : self.onText
   }

   try:
      while reader.Read():
         nodeType = reader.NodeType
         handler = nodeTypeHandlers.get(nodeType)
         if handler:
            handler(reader)
         else:
            raise XmlException(
               "invalid data at line %d" %
               reader.LineNumber)
   finally:
        reader.Close()
```

Tracks which element you're in (`self._currentElement = None`)

Maps node types to handlers (`nodeTypeHandlers = {`)

Loops through document (`while reader.Read():`)

Calls relevant node handler (`handler(reader)`)

`nodeTypeHandlers` is a dictionary mapping the different `XmlNodeTypes` to handler functions. The call to `reader.Read()` advances the reader to the next node. It returns `True` if the reader finds a node, or `False` when the end of the document is read—which ends the reading loop.

`nodeTypeHandlers.get(reader.NodeType)` looks up the current node in the dictionary of handlers. If the node type isn't recognized, then `get` returns `None` and an `XmlException`[4] is raised. If the node type *is* recognized, then the handler is called and the reader passed in. Figure 5.2 shows how the read loop maps XML components it encounters into the handler method calls.

The `finally` block around the read loop ensures that the reader is closed, whatever error might occur.

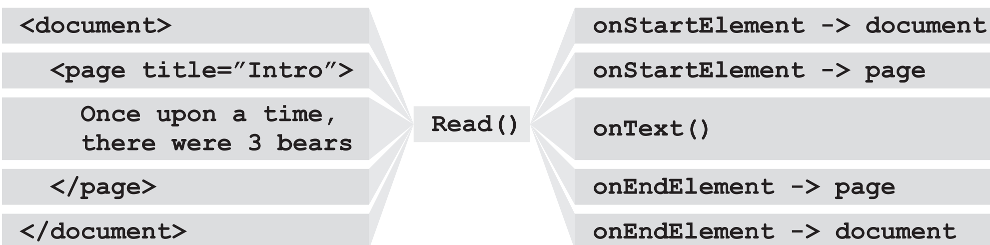


Figure 5.2 The mapping of an XML document to node handler calls

[4] From the `System.Xml` namespace.

The first node handler is `onStartElement`. This will be called when the reader encounters a start element tag such as `<document>` or `<page title="Page title">`. `onStartElement` is shown in listing 5.5.

Listing 5.5 Node handler for start element tags

```
def onStartElement(self, reader):
    name = reader.Name                          # Current element name
    self._currentElement = name

    attributes = {}
    while reader.MoveToNextAttribute()          # Reads all attributes from element
        attributes[reader.Name] = reader.Value

        startHandler = self._elementHandlers.get(name,
                            MISSING_HANDLERS)[0]        # Finds right element handler
        if startHandler:
            startHandler(reader.LineNumber, attributes)
        else:
            raise XmlException("invalid data at line %d" %
                            reader.LineNumber)
```

`onStartElement` needs to extract the attributes from the element. These are collected in the `attributes` dictionary in another loop that calls `reader.MoveToNextAttribute()`. Next you check in the `_elementHandlers` to see if you have a handler for this type of element. If you do, the handler is called with the line number and attributes as arguments. Otherwise, an `XmlException` is raised.

`_elementHandlers` is a dictionary, and `get` returns a default value if the key (the element name) is missing. You use `MISSING_HANDLERS` as the default value (as defined in listing 5.1). Every element needs a handler for the start tag, end tag, and contents.

Element handlers are passed into the constructor as a dictionary that maps element names to the three handler functions. Elements can be self-closing (like `<element />`), which means that they won't have any contents or a separate end tag. The only *required* handler is the start tag.[5] The three handlers for each element type should be provided in a tuple (which will become clearer later when you use `XmlDocumentReader` to read MultiDoc documents). For handling the page element, you have these handlers:

```
elementHandlers = {
    'page': (handleStartPage, handlePageText, handleEndPage)
}
```

If you have elements that don't have text contents or don't need their end tag handling, then you can replace these handlers with `None`.

```
elementHandlers = {
    'someElement': (handleStartPage, None, None)
}
```

`MISSING_HANDLERS` is a tuple of three `None`s. `self._elementHandlers.get(name, MISSING_HANDLERS)` will *always* return a tuple of three values. The start tag handler is

[5] In fact, for self-closing elements, the `XmlReader` *doesn't* call the end tag handler.

the first value; if the element isn't contained in the `elementHandlers` dictionary, then this value will be `None` and an exception will be raised.

The middle handler in the tuple, index position one, is the handler for the text contents of elements. Listing 5.6 shows the code for the `onText` method. It's similar to `onStartElement`, but simpler, because it doesn't need to collect attributes. Because the text handler is optional, it doesn't throw an exception if the text handler is `None`. The text handler gets called with the same arguments as the start tag handler.

Listing 5.6 Node handler for element text values

```
def onText(self, reader):
   element = self._currentElement
   textHandler = self.__elementHandlers.get(element,
                  MISSING_HANDLERS)[1]           <-- Fetches handler
   if textHandler:
      textHandler(reader.LineNumber, reader.Value)     <-- Calls handler
```

Tuples of element handlers have three members. The third member is the handler for end element tags. Listing 5.7 shows `onEndElement`.

Listing 5.7 Node handler for end element tags

```
def onEndElement(self, reader):
   endHandler = self._elementHandlers.get(reader.Name,
                  MISSING_HANDLERS)[2]
   if endHandler:
    endHandler(reader.LineNumber)
```

The end tag handler is called with the line number.

In the first part of the chapter, we looked at first-class functions in Python. In this part, we've put together a general-purpose XML reader class that uses handler functions stored in a dictionary to process different nodes. As long as the handler functions all have the same signature (by taking the same arguments), you're free to implement your handler functions how you want.

To illustrate our general purpose reader at work, let's implement the handler functions needed for the MultiDoc XML save format.

5.4 Handler functions for MultiDoc XML

MultiDoc documents are represented programmatically by the classes `Document` and `Page`. To read them in, you can use `XmlDocumentReader` to read the XML and re-inflate the model classes.

The stages of reading a saved document are shown in figure 5.3.

Steps 2 through 4 will obviously be repeated for every page in the document.

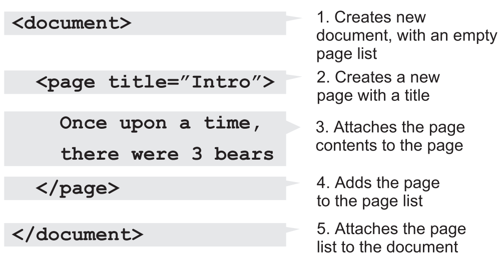


Figure 5.3 Turning a MultiDoc XML document into model class instances

To use XmlDocumentReader, you need to provide handler functions for these steps; the handlers map to the steps as follows:

1. Document start element handler
2. Page start element hander
3. Page text handler
4. Page end element handler
5. Document end element handler

Before reading the file, you need to set up some state that will be used in the reading process. You need to store the document when it's first created, a list to read pages into, and the filename to read. You also need to keep track of the current page so that the text handler can attach the text to it.

The document reader needs access to the model classes and the XmlDocument-Reader. It will also do some verifying of the document structure, so you should import XmlException. DocumentReader sounds like a reasonable name for a document reader class. Listing 5.8 shows the imports and constructor for DocumentReader.

Listing 5.8 Initializing the DocumentReader

```
from model import Document, Page
from xmldocumentreader import XmlDocumentReader, XmlException

class DocumentReader(object):
   def __init__(self, fileName):
      self.fileName = fileName
      self.document = None
      self.currentPage = None
      self.pages = []
```

Because we spent some time creating and discussing XmlDocumentReader, it ought to be easy to use. Listing 5.9 demonstrates just how easy.

Listing 5.9 Setting up the handlers and calling XmlDocumentReader

```
def read(self):
   handlers = {                                      <-- Element handlers for MultiDoc
      'document': (self.onStartDocument,
                   None,
                   self.onEndDocument),
      'page': (self.onStartPage,
               self.onPageText,
               self.onEndPage)
   }
   self.reader = XmlDocumentReader(handlers)         <-- Initializes XmlDocumentReader with handlers
   self.reader.read(self.fileName)
   return self.document                              <-- Returns freshly created document
```

We've identified five handlers you need to pass in. Each element needs a tuple of three handlers: start tag handler, text handler, and end tag handler. MultiDoc documents have a humble two tags: document and page. The document element has child

page tags, but no content; it doesn't need a text handler, and the middle element can be `None`. The `read` method of `DocumentReader` sets up these handlers, using yet-to-be-defined methods. When `XmlDocumentReader` works its magic, the handlers are called—which builds the document.

The important pieces of the jigsaw puzzle are the handlers themselves. Listing 5.10 contains the document start and end tag handlers.

Listing 5.10 Handlers for the `document` element

```
def onStartDocument(self, lineNumber, attributes):
   self.document = Document(self.fileName)

def onEndDocument(self, lineNumber):
   self.document.pages = self.pages
```

Both handles are simple. When you start reading the document (or, encounter the document start tag), you create a new document with the right filename. By the time you encounter the document end tag, you should have read all the pages, and `onEndDocument` should attach the pages to the document. `XmlDocumentReader` can then complete and `read` return the document.

Reading pages requires three handlers; the code for these is in listing 5.11.

Listing 5.11 Handlers for the `page` element

```
def onStartPage(self, lineNumber, attributes):
   title = attributes.get('title')
   if title is None:
      raise XmlException('Invalid data at line %d' %
                    lineNumber)
   self.currentPage = Page(title)

def onPageText(self, lineNumber, value):
   self.currentPage.text = value.replace('\n', '\r\n')

def onEndPage(self, lineNumber):
   self.pages.append(self.currentPage)
```

Pages need a title, which is extracted from the attributes in `onStartPage`. If the title is missing, then the document is invalid, and an exception is raised. If a title is present, then a new page is created and set as the current page.

When the page contents are read in, the page is set as the current page in `onPageText`. `XmlDocumentReader` reads in files using an `XmlReader` returned by `XmlReader.Create`. `XmlReader.Create` returns an instance of `XmlTextReader`, which unsurprisingly opens files for reading in text mode. Python tries to do you a favor by converting `\r\n` line-endings into `\n` when you read files in text mode, which normally enables you to ignore cross-platform differences when dealing with line endings. Unfortunately, this favor backfires when you need to set the text on a Windows Forms control, which doesn't recognize `\n` as a line ending. To avoid this problem, `onPageText` converts `\n` into `\r\n` in the value passed to it.

When you reach the end page element, onEndPage adds the current page to the page list.

It may seem like there isn't a lot of checking to ensure that the document is valid, beyond a cursory check that the title is present. In fact, there's quite a lot of checking being done for you. Any unrecognized nodes or elements would cause an XmlException to be raised from inside XmlDocumentReader. If the XML document itself is invalid (due to missing or misplaced tags, for example), then the .NET XmlReader will raise an error. It will be important for you to catch these potential errors and alert the user when you read in documents.

DocumentReader is a concrete example of using XmlDocumentReader. DocumentReader is a simple class, only 44 lines of Python code, but extending it to read more complex documents and construct more intricate objects from them should be easy.

Now that you've created these classes to form MultiDoc document objects from saved files, you need to plug them into the user interface. To do this, you need an open command.

5.5 The Open command

The OpenCommand is going to provide the mirror functionality to the SaveCommand and will look similar.[6] Instead of the SaveFileDialog, it will use its close (but equally familiar) cousin, the OpenFileDialog (figure 5.4), so that the user can choose a file to open. The open command is *very* similar to the save commands. The similarity even extends to the imports—except that, obviously, OpenFileDialog is imported instead of SaveFileDialog. OpenCommand also needs access to the DocumentReader class and

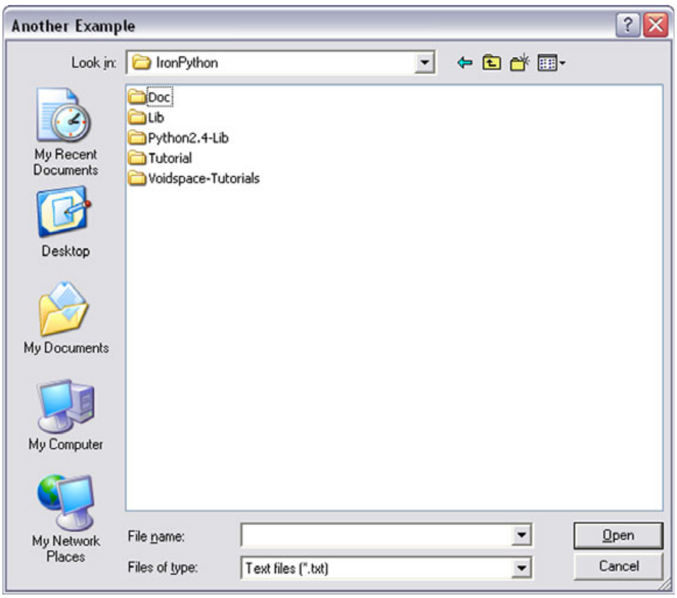


Figure 5.4 The OpenFileDialog in action

[6] But, annoyingly, it's different enough that little code that can be shared.

Xml-Exception. If any errors are raised while reading a file, an XmlException will be raised. To catch these errors, you need to import XmlException.

Listing 5.12 is the imports and initialize for OpenCommand. Like SaveCommand, you have a title and a filter for use on the dialog control.

Listing 5.12 Initializing the OpenCommand

```
from System.IO import Directory, Path
from System.Windows.Forms import (
   DialogResult, MessageBox,
   MessageBoxButtons, MessageBoxIcon,
   OpenFileDialog
)

from documentreader import DocumentReader, XmlException
from savecommands import filter

class OpenCommand(object):

   title = "Open Document"

   def __init__(self, mainForm):
      self.openFileDialog = OpenFileDialog()
      self.mainForm = mainForm

      self.openFileDialog.Filter = filter
      self.openFileDialog.Title = self.title
```

There are two things of note in this otherwise unremarkable code. The first is the reuse of the filter from the save commands. When you switched to an XML file format, you *could* have switched the save file extension too; personally, I (Michael) like .txt because it makes the files easier to open with a text editor. The second thing is that the OpenCommand constructor needs the MainForm passed in. When a new document is created, it needs to be set back on MainForm.

These commands need to implement an execute method. For OpenCommand, this is the method that asks the user to choose a file and attempts to open it as a MultiDoc file. You do the same manipulation with the filename (and directory) as you did for the save commands. If the current document already has a filename, then you set the initial directory and filename on the dialog box (listing 5.13).

Listing 5.13 execute method of OpenCommand

```
def execute(self):
   fileName = self.mainForm.document.fileName
   directory = Path.GetDirectoryName(fileName)
   directoryExists = Directory.Exists(directory)
   openFileDialog = self.openFileDialog

   if fileName is not None and directoryExists:
      openFileDialog.InitialDirectory = directory
      openFileDialog.FileName = fileName

   if openFileDialog.ShowDialog() == DialogResult.OK:     # Asks user for filename
      document = self.getDocument(openFileDialog.FileName)     # Attempts to load document
```

```
    if document:
        self.mainForm.document = document
```
Sets document on form

If the call to openFileDialog.ShowDialog() returns DialogResult.OK, then the user has selected a file, and self.getDocument is called with the filename. Reading the document *could* result in an exception being thrown if the document is invalid. If an exception is raised, it's trapped inside getDocument, and a MessageBox is displayed to the user before None is returned. If a real document is returned, then it's set on the MainForm.

Listing 5.14 shows the implementation of getDocument.

Listing 5.14 The getDocument method of OpenCommand

```
def getDocument(self, fileName):
    try:
        reader = DocumentReader(fileName)
        return reader.read()
    except (IOError, XmlException), e:
        name = Path.GetFileName(fileName)
        MessageBox.Show(
            'Could not read file "%s"\r\nThe error was:\r\n%s' %
            (name, str(e)),
            "Error Saving File",
            MessageBoxButtons.OK,
            MessageBoxIcon.Error
        )
        return None
```

getDocument is almost identical to the saveFile method of SaveCommand, except for the following three differences:

- It uses DocumentReader instead of DocumentWriter.
- It returns a new document object or None, instead of True or False.
- It traps for a tuple of exceptions (IOError, XmlException).

Two possible errors can occur. The first is that you fail to read the file from disk, perhaps due to a hard drive failure or the user whipping out a USB stick in between selecting the file with the dialog box and you actually managing to read it. This would cause an IOError. Alternatively, the document could be badly structured, resulting in an XmlException. If either situation happens, then you trap the error and alert the user with a message box.

At this point, we'd *love* to say that our work is done. Unfortunately, there's a problem with this implementation of OpenCommand.

When the new document is returned from getDocument, it's set on the MainForm. It is not just the MainForm that holds a reference to the current document, but also the TabController and the save commands. The TabController also needs to take *action* when a new document is loaded—it needs to update the tabs to reflect the new document.

To solve this problem, chapter 6 will set up a document observer system. But before we get to that, let's review what we've done in this chapter.

5.6 Summary

We started the chapter by looking at first-class functions in Python. Being able to treat functions (or other callable objects) as ordinary objects enables some useful patterns. There's a lot more to functional programming than we've covered here. Functional programming languages have been used mainly in academia, but they've been enjoying something of a renaissance recently. Languages like Haskell and Erlang are starting to gain popularity, but they're still seen as harder to learn than imperative languages. Python supports functional programming, without limiting you to one particular style.

The rest of the chapter was concerned with XML reading and writing for MultiDoc. The approach we showed for reading should handle enough of the XML components for reading complex documents. It should also be easily extensible (simply add more node type handlers) for supporting parts of the XML spec that we haven't covered. The `XmlDocumentReader` class demonstrates the power of IronPython. The .NET framework provides powerful and effective ways of dealing with XML that can be used from IronPython with much less code (which means more readable code) than from other .NET languages.

We did a small refactoring of the `SaveCommand` to use the XML writing capabilities, but we haven't yet extended the MultiDoc user interface to incorporate the new `OpenCommand`.

The next chapter creates the additional features that MultiDoc is still missing, and integrates them into the user interface. This is the exciting part; you get to transform MultiDoc from the bare shell it is now (with exposed wires and plumbing—to return to our building analogy) into a functioning application. But you'll need to get a little assistance from Visual Studio.

Properties, dialogs, and Visual Studio

This chapter covers

- Properties in Python
- The observer pattern
- Windows Forms dialogs
- Using Visual Studio with IronPython
- Object serialization

In the last chapter, you added to the MultiDoc codebase the ability to load and save multiple pages with XML. Alas, we didn't have the chance to give the user access to your work. In this chapter, we bring life to the skeleton of the MultiDoc framework by clothing it with the flesh of additional features and functionality. Some of these features will require the refactoring of classes you've already written. In the process, you'll learn more about Python and get to use Visual Studio with IronPython plus some .NET classes that you haven't seen yet.

Although you've extended MultiDoc to read and write XML files, you still don't have a way of updating all the parts of the code that need access to documents. It's the responsibility of the tab controller to keep the view and the model synchronized;

it needs a way of updating the view when the model is changed. The save commands also need a reference to the current document. You can provide this by making some classes *document observers.*

6.1 Document observers

The model classes should remain loosely coupled to the view and controllers (a substantial part of the Model-View-Controller pattern), so you can't give the document the responsibility for telling the tab controller to update the view.

Now that `OpenCommand` can create new documents, we want the following to happen:

1. `OpenCommand` to load and set the document on the main form
2. The save commands to update with the new document
3. The tab controller to update with the new document
4. The tab controller to update the view appropriately

We implement all these steps in this section.

For the main form to know which objects to update, it needs a list of objects interested in the document—these are the document observers. Implementing a list of objects that need to keep track of the document will get you half way toward a pattern known as the *observer pattern.*[1] The observer pattern is a common pattern to use along with Model-View-Controller.

The full observer pattern involves providing a mechanism (usually an event mechanism) for model classes to inform controllers that a change has occurred. You don't yet need this part of the pattern, but the first step of the pattern is to have a central mechanism for keeping all observers updated with the current model classes. In our case, the relevant model class is the document.

In the tail end of the last chapter, you implemented code in the `OpenCommand` that sets the document attribute on the main form. When this happens, you need the aforementioned behavior to be triggered. Attributes that trigger behavior when they're set are known as *properties.* Python also has properties, so this seems like an appropriate place to introduce them.

6.1.1 Python properties

Properties, which are common in other languages including C#, are a mechanism for giving you control over what happens when an attribute is accessed, set, or deleted. We've already used them a great deal when configuring and using .NET classes.

They're an alternative to providing get/set/delete methods. They don't do anything that you couldn't achieve with getter and setter methods, but they allow the external API of an object to be more elegant and natural to use. They also allow you to seamlessly migrate a Python API from using normal attributes to providing custom behavior. This is the process we're going through now with the document.

[1] See http://en.wikipedia.org/wiki/Observer_pattern.

Attribute access and descriptors

In Python you can *completely* control the way attributes are accessed through the descriptor protocol.[2] Under the hood, descriptors are how bound methods on instances get called with `self`, and lots more.

Properties are one convenient use of the descriptor protocol. Fortunately, you don't need to understand descriptors in order to use properties.

Python properties are created using the built-in type `property`. It behaves like a function, but is used inside a class namespace to bind methods that you want to be called when an attribute is accessed.

The full signature of `property` is as follows:

```
name = property(fget=None, fset=None, fdel=None, doc=None)
```

- *`fget`*—A method to be called when the attribute is fetched
- *`fset`*—A method to be called when the attribute is set
- *`fdel`*—A method to be called when the attribute is deleted
- *`doc`*—A docstring for the property

All four arguments are optional (and can be passed in as keyword arguments if you want), so you only need to provide the methods that you need. (It's relatively unusual to provide a docstring for properties, unless you're using an automated documentation generating tool.)

Listing 6.1 is an example of a class with a property `x`, with custom behavior when the property is fetched, set, or deleted. The underlying value for `x` is stored in an attribute called `_x`.

Listing 6.1 An example of a property with methods for fetching, setting, and deleting it

```
class PropertyExample(object):
   def __init__(self, x):          <- Initializes an instance
      self._x = x                     with a value

   def _getX(self):
      print "getting x"
      return self._x

   def _setX(self, value):
      print "setting x"
      self._x = value

   def _delX(self):
      print "Attempting to delete x"

   x = property(_getX, _setX, _delX)     <- Creates property
```

[2] See http://users.rcn.com/python/download/Descriptor.htm.

The following interactive interpreter session demonstrates it in use:

```
>>> example = PropertyExample(3)
>>> example.x
getting x
3
>>> example.x = 6
setting x
>>> del example.x
Attempting to delete x
>>> example.x
getting x
6
```

Even after attempting to delete it, the `x` attribute is still available. You could have similar behavior just by omitting the `fdel` argument (the third argument) to `property`; attempting to delete `x` would then raise the exception `AttributeError: undeletable attribute`. Similarly, omitting the `fset` argument (the second argument) would create a read-only property that can't be set.

So how do you use properties to implement document observers on the `MainForm`?

DOCUMENT OBSERVERS ON THE MAINFORM

Now that you understand properties, you can make the document a property on `MainForm`. We've already outlined what we want to achieve. Setting the document on `MainForm` should update all the observers and cause the tab controller to update the view.

You can do this in the following three steps:

1. Make document a property on `MainForm` and have it maintain a list of document observers.
2. Make document a property on the tab controller, and have it update the view when a new document is set.
3. Add the open command to the menu and toolbar.

MAKING DOCUMENT A PROPERTY

The objects that need to keep track of the current document are the two save commands and the tab controller. Listing 6.2 creates a list of the observers and makes `document` a property.

Listing 6.2 Creating a `document` property on `MainForm`

```
def initializeObservers(self):
   self.observers = [
      self.saveCommand,
      self.saveAsCommand,
      self.tabController
   ]

def _setDocument(self, document):
   self._document = document
   for observer in self.observers:
      observer.document = document

document = property(lambda self: self._document, _setDocument)
```

The first argument to `property` is the getter method. This is a lambda that returns `_document`, the underlying instance variable used to store the real document. `_setDocument` is the setter method. When a new document is set, it iterates over all the observers and sets the document on them. Note that even when you add the `OpenCommand` to `MainForm`, it won't need to be added to the list of observers.[3]

You also need to change the `__init__` method of `MainForm` to use the new observer system. You remove the line `doc = self.document = Document()` and change the last part of `__init__` to

```
self.tabController = TabController(tab)

self.initializeCommands()
self.initializeToolbar()
self.initializeMenus()
self.initializeObservers()
self.document = Document()
```

This creates the tab controller without passing the document, and then it initializes the list of observers and sets a default, empty document. Setting the document triggers the property to set the document on all the observers. (The document should also no longer be passed into the save commands in `initializeCommands`.)[4]

The next thing to do is to refactor the tab controller to also use a `document` property.

REFACTORING THE TABCONTROLLER

Currently the `TabController` is initialized with a document and creates a tab for every page. It no longer needs to be given the document when it's initialized, but it needs to create the tabs whenever a document is set. If a document is already open, then the existing tabs need to be removed first.

Luckily, this is easy to achieve by moving the code that creates the tab pages from the `TabController` constructor and into the setter method for the `document` property. You also need another small piece of magic; all the changes necessary are shown in listing 6.3.

Listing 6.3 Changing `TabController` to have `document` property

```
class TabController(object):

   def __init__(self, tabControl):          # Changed to not take document
      self.tabControl = tabControl
      self._document = None

   def _setDocument(self, document):
      if self._document is not None:
         self.tabControl.SelectedIndexChanged -= self.maintainIndex   # Removes SelectedIndexChanged event handler

      self._document = document
      self.tabControl.TabPages.Clear()      # Clears all tab pages
```

[3] It only uses the document to check if it has a filename, and it can get this via the reference to `MainForm` that it keeps.

[4] This Herculean task we leave to you—alternatively, you can download the source code.

```
        for page in document.pages:
            self.addTabPage(page.title, page.text)

        self.index = self.tabControl.SelectedIndex
        if self.index == -1:
            self.index = self.tabControl.SelectedIndex = 0
        self.tabControl.SelectedIndexChanged += self.maintainIndex

    document = property(lambda self: self._document, _setDocument)
```

Adds event handler again

When a new document is set on the tab controller, any current tab pages are cleared with a call to `tabControl.TabPages.Clear()`. Clearing the tabs can cause the `SelectedIndexChanged` event to be raised several times. To avoid this, if you already have a document set, you first remove the event handler. After creating new tab pages for the document, you wire up the event again.

The last step in finally getting the open command to work is adding a menu item and toolbar button to the view that launches it.

6.1.2 Adding the OpenCommand

The pattern for creating new menu items and new toolbar buttons is well established in `MainForm`. Adding a new Open item, to both menu and toolbar, can be achieved with a little copy-and-paste magic and another icon from those wonderful folk at glyFx. When we eventually get the `OpenCommand` wired in, it will look like figure 6.1.

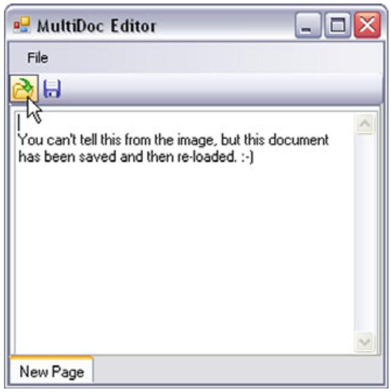


Figure 6.1 MultiDoc Editor with the `OpenCommand` in place

The first change is to add the `OpenCommand` to the `initializeCommands` method. The `OpenCommand` is initialized with a reference to the `MainForm` instance. Because this method is being *called* by the `MainForm`, you pass in `self` to the `OpenCommand` constructor.

```
def initializeCommands(self):
    tabC = self.tabController
    self.saveCommand = SaveCommand(tabC)
    self.saveAsCommand = SaveAsCommand(tabC)
    self.openCommand = OpenCommand(self)
```

As you can see from this segment, the save commands only receive the tab controller and not the document.

Once you've put this code in place, MultiDoc will be capable of loading the new documents saved in XML format. This still doesn't meet our original specification because you have no way of *creating* new pages.

The next section will address this; along the way you'll learn about dialogs and using Visual Studio Express with IronPython.

6.2 More with TabPages: dialogs and Visual Studio

In our exploration of creating an application with IronPython, we've managed to work our way through about three-quarters of our original specification in only 520 lines of pure-Python code. We've divided MultiDoc into eight modules, and it's proving easy to refactor and extend. Importantly, the code is also *readable*.

The last piece of functionality to make MultiDoc *useful* is the ability to add new tab pages. Because our tab pages have names, you also need to be able to specify the name when you create tab pages, and preferably rename them later. While we're at it, it would be nice to be able to delete pages as well. This is the easiest code to write and prepares the way nicely for learning about creating dialogs.

6.2.1 Remove pages: OK and Cancel dialog box

When the user asks to perform an irreversible action, such as removing a tab page, it's normal to ask for confirmation of the action. You're undoubtedly familiar with the standard Windows OK/Cancel dialog box, which gives the user the choice of whether to continue with the action or to abort.

This is actually our old friend the system message box.

In case you've forgotten, the most useful overload of `Show` is as follows:

```
MessageBox.Show(String, String,
   MessageBoxButtons, MessageBoxIcon)
```

The two strings are the text and caption (body and title) of the message box.

When you used the message box to alert the user of errors loading or saving a file, you only had an OK button. You specified this by passing in `MessageBoxButtons.OK`.

You can create a message box with different buttons by providing a different member of the `MessageBoxButtons` enumeration in the call to `Show`. The possible options are listed in table 6.1.

Table 6.1 Members of the `MessageBoxButtons` enumeration for specifying the buttons on the system message

`MessageBoxButtons` enumeration member	Effect
`AbortRetryIgnore`	The message box contains Abort, Retry, and Ignore buttons.
`OK`	The message box contains an OK button.
`OKCancel`	The message box contains OK and Cancel buttons.
`RetryCancel`	The message box contains Retry and Cancel buttons.
`YesNo`	The message box contains Yes and No buttons.
`YesNoCancel`	The message box contains Yes, No, and Cancel buttons.

You can tell which button the user selected by the return value of `Show`, which will be a member of the `DialogResult` enumeration. Every possible button has a corresponding member on `DialogResult`. Table 6.2 lists all the members.

Table 6.2 Members of the `DialogResult` enumeration for interpreting the return value of a dialog or message box

`DialogResult` enumeration member	Meaning
`Abort`	The dialog box return value is Abort (usually sent from a button labeled Abort).
`Cancel`	The dialog box return value is Cancel (usually sent from a button labeled Cancel).
`Ignore`	The dialog box return value is Ignore (usually sent from a button labeled Ignore).
`No`	The dialog box return value is No (usually sent from a button labeled No).
`None`	Nothing is returned from the dialog box. This means that the modal dialog box continues running.
`OK`	The dialog box return value is OK (usually sent from a button labeled OK).
`Retry`	The dialog box return value is Retry (usually sent from a button labeled Retry).
`Yes`	The dialog box return value is Yes (usually sent from a button labeled Yes).

For our use case, `MessageBoxButtons.OkCancel` seems like the right choice. Only `MessageBoxIcon` is left to choose; the options are displayed in table 6.3.

Table 6.3 Members of the `MessageBoxIcon` enumeration for specifying the icon in a message box

`MessageBoxIcon` enumeration member	Icon description
`Asterisk`	The message box contains a symbol consisting of a lowercase letter *i* in a circle.
`Error`	The message box contains a symbol consisting of a white X in a circle with a red background.
`Exclamation`	The message box contains a symbol consisting of an exclamation point in a triangle with a yellow background.
`Hand`	The message box contains a symbol consisting of a white X in a circle with a red background.
`Information`	The message box contains a symbol consisting of a lowercase letter *i* in a circle.
`None`	The message box contain no symbols.
`Question`	The message box contains a symbol consisting of a question mark in a circle.
`Stop`	The message box contains a symbol consisting of a white X in a circle with a red background.
`Warning`	The message box contains a symbol consisting of an exclamation point in a triangle with a yellow background.

Because we're asking a question, `MessageBox.Question` seems appropriate. Now that that's settled, you're ready to write `RemoveCommand`. Because you'll have several commands for working with tab pages and they'll all be short, let's put them in a module called `tabcommands.py`.

The full call to `MessageBox.Show` looks like the following:

```
result = MessageBox.Show("Are you sure?", "Delete Page"
   MessageBoxButtons.OKCancel, MessageBoxIcon.Question)
```

`result` will then either be `DialogResult.OK` or `DialogResult.Cancel`. Now that you know how to use the message box, you need to build the infrastructure that can act on the user's choice.

THE CODE BEHIND

To fully implement a `RemovePage` command, there are three distinct steps:

1. Ask the user for confirmation.
2. Remove the currently visible tab.
3. Delete the corresponding page from the model.

Because the first step triggers the following two, you should implement those first. But before that, we need to avoid a potential bug.

There's always the possibility that the user will try and activate the command after there are no pages left. You need to ensure that this situation doesn't cause MultiDoc to crash. It's the controller's job to synchronize the model and the view, so the right place to implement this is in the tab controller.

It would be good to check whether there's at least one tab page *before* displaying the message box. You can tell whether a tab control has any pages in a couple of ways; the number of pages will be 0, and the `SelectedIndex` property will return -1. To check whether there are any pages, you can add a property `hasPages` to the tab controller. This wraps a simple function (a lambda) that checks `SelectedIndex`. It needs to return `False` if the result is -1; otherwise, `True`.

```
hasPages = property(lambda self: self.tabControl.SelectedIndex != -1)
```

To delete the current page, you need to delete the currently selected index from the document's list of pages. You also need to remove the current visible tab page from the `TabPages` collection.

Listing 6.4 shows the implementation of `deletePage`. Even though you intend to check `hasPages` *before* asking the user to confirm page deletion, you should still check inside this method. This method is now part of the public API of the `TabController` and should be safe against being called when there are no pages.

Listing 6.4 Method on `TabController` to delete pages on the view and the model

```
def deletePage(self):
   if not self.hasPages:
      return
   index = self.tabControl.SelectedIndex
```

```
    del self.document[index]
    tabPage = self.tabControl.SelectedTab
    self.tabControl.TabPages.Remove(tabPage)
```

You need to make the same change to the `updateDocument` and `maintainIndex` methods, a process which raises another issue. Deleting a page causes the `SelectedIndex` to change, so `maintainIndex` will be called. This isn't generally a problem; the extra call to `updateDocument` won't usually do any harm. It *is* a problem when the deleted page is the *last* tab page. The index of the deleted page (stored as `self.index`) will no longer be a valid index for the tab pages at all, and `updateDocument` will blow up. `maintainIndex` still needs to update the index tracking variable, but you should make the call to `updateDocument` conditional on the stored index being valid. Listing 6.5 shows the new `maintainIndex`.

Listing 6.5 Keeping the model updated from user input in the view

```
def maintainIndex(self, sender, event):
    if not self.hasPages:
        return
    if self.index < len(self.tabControl.TabPages):
        self.updateDocument()
    self.index = self.tabControl.SelectedIndex
```

Now on to the `RemoveCommand` itself. It needs to be initialized with a reference to the tab controller. In the execute method, it should check whether there are any tab pages. If there are, then it should ask the user to confirm, calling `deletePage` if the user hits OK.

This is a rather short piece of code, as shown in listing 6.6.

Listing 6.6 Asking for confirmation before calling `deletePage` on `TabController`

```
from System.Windows.Forms import (
    DialogResult, MessageBox,
    MessageBoxButtons, MessageBoxIcon
)
class RemoveCommand(object):

    def __init__(self, tabController):
        self.tabController = tabController

    def execute(self):
        if not self.tabController.hasPages:
            return
        result = MessageBox.Show("Are you sure?",
                          "Delete Page",
                          MessageBoxButtons.OKCancel,
                          MessageBoxIcon.Question)
        if result == DialogResult.OK:
            self.tabController.deletePage()
```

The result of activating this command, with at least one page present, is the message box in figure 6.2. At this point, you've successfully completed creating the Delete Page functionality.



Figure 6.2 The Delete Page message box with OK and Cancel buttons

Frequently, presenting the user with an OK/Cancel choice isn't enough. In the next section, we look at creating custom dialogs.

6.2.2 Rename pages: a modal dialog

For the user to provide a new name to rename a page, you need to present some kind of dialog. The right control to enter the new name in is the `TextBox`. By default, the old name should appear in the text box so that the user can edit it rather than having to type the name in full.

Again, you'll need some support for this in the tab controller, both to fetch the current page title and to set the new one. Because you need to fetch and set the current page title, you can make it a property on the tab controller, as in listing 6.7.

Listing 6.7 Providing a `currentPageTitle` for fetching and setting title

```
def _getCurrentPageTitle(self):
   if not self.hasPages:
      return None
   index = self.tabControl.SelectedIndex
   return self.document.pages[index].title

def _setCurrentPageTitle(self, title):
   if not self.hasPages:
      return
   index = self.tabControl.SelectedIndex
   page = self.document.pages[index]
   page.title = title
   self.tabControl.SelectedTab.Text = title

currentPageTitle = property(_getCurrentPageTitle, _setCurrentPageTitle)
```

Dialogs are displayed modally, blocking the GUI until the user completes the action by selecting OK or Cancel (or whatever buttons you provide).

Creating dialogs is simple; they're simply forms that you display with a call to `ShowDialog` instead of `Show`. They can contain any controls you like. There are several other subtleties to making a form into something that looks and feels like a dialog. We'll go through these as we create `RenameTabDialog`.

This dialog will be very simple. All it needs is a text box to edit/enter the name, and an OK and a Cancel button. Because we'll shortly be creating a command for adding tab pages, which will include choosing a name, it makes sense if the dialog is flexible enough for both jobs.

Listing 6.8 is the initialization for the new command.[5]

Listing 6.8 The constructor for `RenameTabDialog`

```
class RenameTabDialog(Form):

   def __init__(self, name, rename):
      title = "Name Tab"                  <-- Dialog title for new pages
```

[5] As with all our examples, see the full version in the downloadable examples for all the imports that go with this code.

```
if rename:
   title = "Rename Tab"
self.Text = title
self.Size = Size(170, 85)
self.FormBorderStyle = FormBorderStyle.FixedDialog
self.ShowInTaskbar = False
self.Padding = Padding(5)

self.initializeTextBox(name)
self.initializeButtons()
```

Dialog title for renaming pages

Makes dialog invisible in the taskbar

As with `MainForm`, the dialog is a subclass of `Form`. It takes two arguments in the constructor: a string `name` (the current name of the tab) and a Boolean `rename` (whether you're renaming a page or creating a new one). The title of the dialog is set differently depending on which of these two actions is taken—which, in fact, is the only difference between the two uses.

This dialog isn't resizable, so you can use `FormBorderStyle.FixedDialog` as the border style. This makes the form non-resizable, and also gives the form an appearance in keeping with a dialog. You set the form to a fixed size, using the `Size` structure from `System.Drawing`. A form with only these options set would look like figure 6.3.

By default, all active forms have an icon in the taskbar. This isn't normal for dialogs, and setting the border style *doesn't*; this is why the dialog sets the `ShowInTaskbar` property to `False`. The code after this sets the padding on the form. (Padding is a structure in `System.Windows.Forms`.) The padding affects the layout of controls that don't have an absolute location specified; this is normally for controls laid out using `Dock`. The layout is done in two methods not yet written: `initializeTextBox`, which needs to know the initial name, and `initializeButtons`.



Figure 6.3 A form with a fixed size and border style set to `FormBorderStyle.FixedDialog`

FILLING IN THE DIALOG

Positioning the text box in the dialog is easy. We want it to be at the top of the dialog (above the buttons), and nearly as wide as the dialog. To position the dialog in the form, you set `Dock = DockStyle.Top`. This will create a text box of default width, which isn't wide enough, so you need to explicitly set the width. You also set the name passed into the constructor as the text in the textbox; you can see this in listing 6.9.

Listing 6.9 Creating and laying out text box for `RenameTabDialog`

```
def initializeTextBox(self, name):
   self.textBox = TextBox()
   self.textBox.Text = name
   self.textBox.Width = 160
   self.Dock = DockStyle.Top

   self.Controls.Add(self.textBox)
```

The next method is to lay out the buttons for the dialog. Unfortunately, getting these positioned correctly is a bit more intricate. The code for this is shown in listing 6.10.

Listing 6.10 Creating, configuring, and laying out buttons for `RenameTabDialog`

```
def initializeButtons(self):
   buttonPanel = Panel()                         <-- Panel to contain buttons
   buttonPanel.Height = 23
   buttonPanel.Dock = DockStyle.Bottom
   buttonPanel.Width = 170

   acceptButton = Button()
   acceptButton.Text = "OK"
   acceptButton.Click += self.onAccept
   acceptButton.Width = 75
   acceptButton.Dock = DockStyle.Left
   acceptButton.DialogResult = DialogResult.OK
   self.AcceptButton = acceptButton              <-- Sets OK as AcceptButton
   buttonPanel.Controls.Add(acceptButton)

   cancelButton = Button()
   cancelButton.Text = "Cancel"
   cancelButton.Width = 75
   cancelButton.Dock = DockStyle.Right
   cancelButton.DialogResult = DialogResult.Cancel
   self.CancelButton = cancelButton              <-- Sets Cancel as CancelButton
   buttonPanel.Controls.Add(cancelButton)

   self.Controls.Add(buttonPanel)
```

The buttons are contained in a panel, which is laid out in the form using `DockStyle.Bottom`. Its width is set to be the same width as the form. Its height is set to 23 pixels, which is the default height of a button. Because we're using `DockStyle.Left` and `DockStyle.Right` to position the buttons in the panel, you need to set an explicit height on the panel, or the buttons will look very odd. You also need to set a width on the buttons, 75 pixels being the default width of a button. Any other combination of settings[6] causes one of these parameters to be overridden.

`initializeButtons` also does some magic which is relevant to creating dialogs. Two default actions are common to most dialogs: accepting and canceling them. These can be triggered by pressing the Esc key (to cancel), or Enter key (to accept). You can tell the form which buttons to treat as accept and cancel buttons with the following two steps:

- Setting the `AcceptButton` and `CancelButton` properties on the form
- Hooking up the appropriate `DialogResult` to the buttons

The accept and cancel actions will be triggered by clicking the buttons *or* by pressing the Enter or Escape keys while the dialog box has focus. The `acceptButton` has an explicit handler, but the default action for the cancel button is fine (returning `DialogResult.Cancel`).

Because we want to just display the dialog and return a result, you can wrap it in a function. With a form displayed as a dialog, accepting or canceling the dialog (even with the close button) doesn't close it; it's merely hidden. You could reuse the dialog;

---

[6] At least all the other myriad combinations we've tried.

but, in this case, it's more convenient to close it so that you don't need to keep track of it and its resources can be freed up. This is exactly what listing 6.11 does.

Listing 6.11 `ShowDialog` function displaying `RenameTabDialog` and returning result

```
def ShowDialog(name, rename):                    <-- Takes same arguments as dialog
   dialog = RenameTabDialog(name, rename)
   result = dialog.ShowDialog()
   dialog.Close()
   if result == DialogResult.OK:
      return dialog.textBox.Text            <-- Sets Return
   return None
```

When `ShowDialog` is called, it displays the dialog. The elegant result can be seen in figure 6.4—note the padding around the buttons! If the user selects OK (or hits the Enter key), then the function returns the text from the text box; otherwise, it returns `None`.



Figure 6.4 The `RenameTabDialog` called from `ShowDialog`

Inspecting the controls after the dialog has returned is one way of retrieving the values the user has selected on the dialog. An alternative approach would be to define (and wire up to the buttons) `onAccept` and `onClose` methods, which can set attributes or data structures on the dialog representing the user choices.

THE RENAMECOMMAND

With the dialog in place, you need a command to use it. This command can go in `tabcommands.py`, and is also very simple—as you can see in listing 6.12. It uses the `currentPageTitle` property you added to the tab controller and the `ShowDialog` function.

Listing 6.12 `RenameCommand`: using the dialog

```
from renamedialog import ShowDialog

class RenameCommand(object):
   def __init__(self, tabController):
      self.tabController = tabController

   def execute(self):
      if not self.tabController.hasPages:
         return
      currentTitle = self.tabController.currentPageTitle

      newTitle = ShowDialog(currentTitle, True)
      if newTitle is not None:
         self.tabController.currentPageTitle = newTitle
```

Getting even this simple dialog to look right is much harder than it should be. GUI layout can be very fiddly, and finding the right combination of controls and layout options takes a lot of experimentation.

One thing that helps is making the `renamedialog` module executable—calling the `ShowDialog` function when it's run directly with IronPython. If you remember from the Python tutorial, you can do this with a conditional block that checks the magic

variable `__name__`. Listing 6.13 shows the conditional block at the start of the import code and then the class and function definitions.

Listing 6.13 Importing code and function calls for module as script

```
if __name__ == '__main__':
   import clr
   clr.AddReference("System.Windows.Forms")
   clr.AddReference("System.Drawing")

from System.Drawing import Size
from System.Windows.Forms import (
   Button, DialogResult,
   DockStyle, Panel, Form,
   FormBorderStyle, Padding, TextBox
)

[RenameTabDialog and ShowDialog...]

if __name__ == '__main__':
   print ShowDialog("Something", False)
   print ShowDialog("Something Else", True)
```

If the script is run as the main script, then `ShowDialog` is called twice, printing the return value. It's called once as a Name dialog and once as a Rename dialog. If the dialog is imported rather than being run, then this code has no effect.

Being able to make minor modifications to a single element of an application and then immediately rerun the script[7] with no compile phase is one of the advantages of working with IronPython. This kind of manual testing is no replacement for an automated test suite, but unfortunately test suites aren't (yet) clever enough to make aesthetic judgments about the appearance of GUIs.

By now .NET developers are probably jumping up and down and crying that there's a much easier way of resolving GUI layout issues. They'd be right; that easier way is Visual Studio. Visual Studio contains a built-in forms designer that lets you do GUI layout with drag-and-drop actions.

So if we have this amazing tool available, why have we put so much effort into manual layout? Visual Studio is ideal for creating fixed-size dialog; for forms that need to cope with resizing, we tend to prefer hand-coded solutions. Obviously, if you don't understand how to lay out forms in code, this won't seem like an option for you. But even with Visual Studio, you have to set properties to the appropriate values, so you need to know what they are and what they mean. More importantly, if something goes wrong or doesn't look right, the only way you'll be able to fix it (or even understand the problem) is by knowing what the designer is doing when you lay out controls.[8]

You now have a working implementation of a manually created Rename Page dialog. This nearly completes the functionality we aimed to add to MultiDoc at the start of the chapter. Having wrestled with manual layout (or at least seen the result of our

---

[7] Here we're running a class library as an executable, for testing purposes.

[8] For a better explanation of this, read Joel Spolsky's *The Law of Leaky Abstractions* at http://www.joelonsoftware.com/articles/LeakyAbstractions.html.

wrestlings), you now get to use Visual Studio. This works well with IronPython; we're sure this will come as a relief if you're a .NET developer, and perhaps something of a pleasant surprise if you're a Python developer.

6.2.3 Visual Studio Express and IronPython

For this example, we use the free version of Visual Studio, Visual Studio Express. Visual Studio Express doesn't have IronPython integration, so we have to find another way to work with it to create our dialog.

Depending on which version you download,[9] it can generate Visual Basic, C#, C++, or Visual J#. Because they all compile to .NET assemblies and we won't be directly writing code in this example, it doesn't really matter which one you choose. Having generated code in another language can be an advantage; it means there's less temptation to fiddle with it!

You can create the dialog layout in Visual Studio, subclass it in IronPython, and program all the *behavior* in the subclass. You need to create an assembly containing a suitable base class for our dialog.

Our base class dialog will be a class library, so open Visual Studio Express and create a new class library. You'll be presented with the interface in figure 6.5.

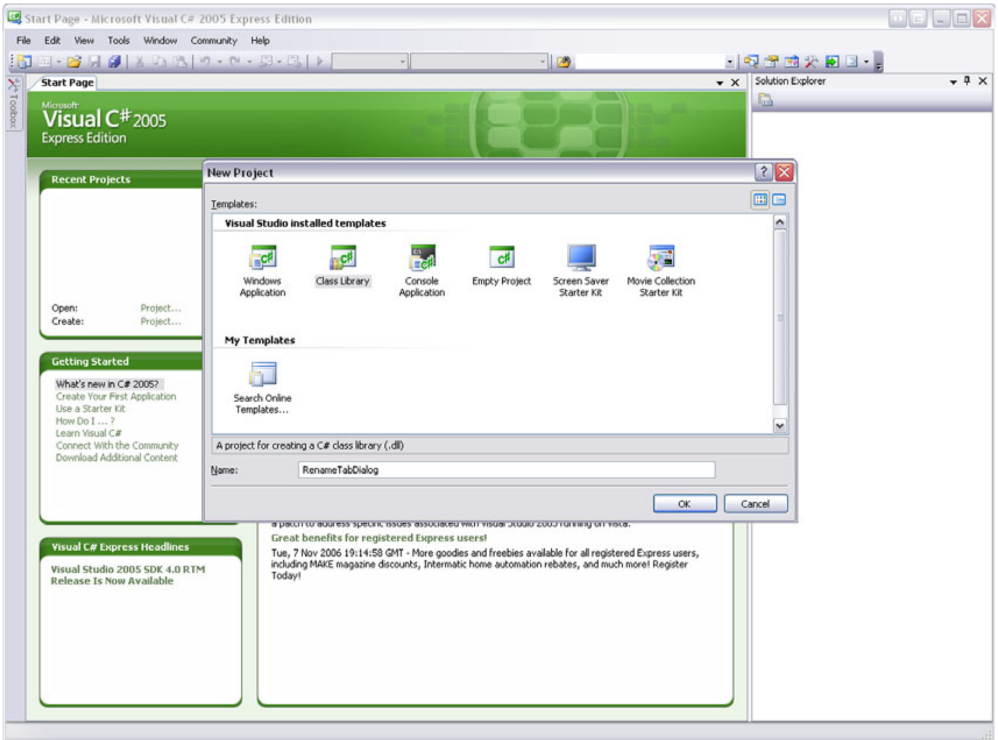


Figure 6.5 Creating a `RenameTabDialog` class library in Visual Studio Express

---

9 See http://msdn.microsoft.com/vstudio/express/. Note that you *can* install multiple versions side by side.

After creating the class library, preferably with a sensible name, you need to add a form to it. If you right-click the project in the Solution Explorer, you should see the menus shown in figure 6.6, and be able to add a form.

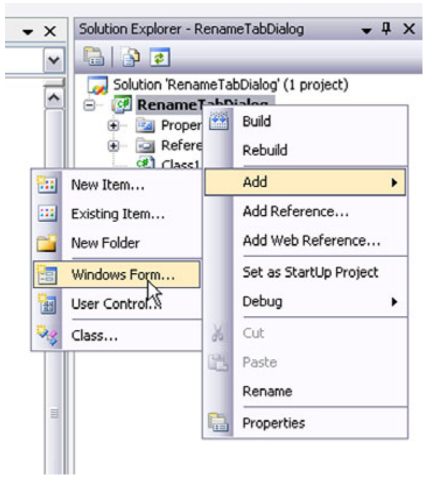


Figure 6.6 Adding a form to the new class library

This will add the form to the project, and Visual Studio will whirr away for a few seconds while it adds references to relevant assemblies such as `System.Windows.Forms`. You can then resize the form and drag controls from the toolbox onto it. You'll also need to open the Properties pane by right-clicking the form and selecting Properties. You want to end up with an interface looking like figure 6.7.

By clicking each element (the form, the buttons, and the text box), you can configure the different properties for each control. For our dialog, you need to complete the following steps:

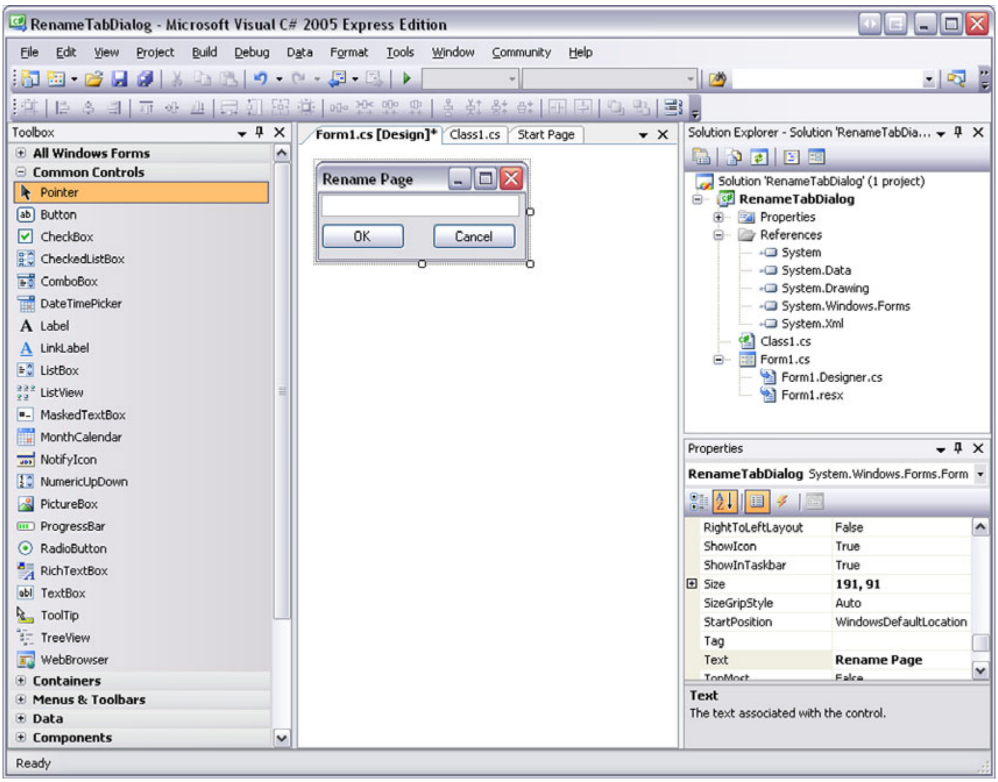


Figure 6.7 Designing the `RenameTabDialog` with the Visual Studio forms designer

1. Name the buttons and set their text.
2. Set the `DialogResult` properties on the buttons.
3. Make sure you name the `TextBox` *textBox* and set the modifier to Public rather than Private. You'll be able to access the property from the `ShowDialog` function.
4. Set the form border to `FixedDialog`.
5. Set the form name to `RenameTabDialogBase`.
6. Set the form `ShowInTaskbar` property to `False`.
7. Set the `AcceptButton` and `CancelButton` properties on the form to `okButton` and `cancelButton`.

Creating the dialog with Visual Studio takes considerably less time than trying to tweak the GUI by hand, but the results are almost identical.

NOTE In IronPython 1, you can access protected .NET members as if they were public.[10] The IronPython 2 rules about accessing .NET members are closer to C#.[11] You'll subclass the dialog we're creating, so protected status would allow you access from inside the subclass—but you also need external access so the text box must be public.

Pressing F6 in Visual Studio (or selecting the Build > Build Solution menu item) compiles our project into an assembly. The compiled assembly will then be available in the bin\Release folder of the project as RenameTabDialog.dll (assuming you named your project RenameTabDialog).

For you to use this assembly from IronPython, it needs to be somewhere on sys.path. The simplest solution is to put it in the same directory as the IronPython file using it. We can then add a reference to RenameTabDialog and import `RenameTabDialogBase` from the `RenameTabDialog` namespace. You'll need to tweak these names to match the ones you've used in the Visual Studio project.

Having done the layout and configuring in the Visual Studio project, we can reduce the amount of code in the dialog. Instead of subclassing Form, you can now create a subclass of `RenameTabDialogBase`. You still need to set the title and the initial text in the text box. The full code for the dialog is shown in listing 6.14.

Listing 6.14 Using the dialog created in Visual Studio with IronPython

```
if __name__ == '__main__':
   import clr
   clr.AddReference('RenameTabDialog')
   clr.AddReference('System.Windows.Forms')

from RenameTabDialog import RenameTabDialogBase
```

10 IronPython uses reflection to discover members. Marking them as protected or private only makes them less convenient to access; it doesn't truly hide them from the determined programmer.

11 If you subclass a .NET class, protected members are public, so you can still access them from outside the class. You can't access protected members on a non-Python subclass.

```
from System.Windows.Forms import DialogResult

class RenameTabDialog(RenameTabDialogBase):
   def __init__(self, name, rename):
      RenameTabDialogBase.__init__(self)

      title = "Name Tab"
      if rename:
         title = "Rename Tab"
      self.Text = title

      self.textBox.Text = name
```

As the external API is the same as the hand-coded dialog, the `ShowDialog` function can remain unchanged.

Although this example of using Visual Studio used a dialog created with the forms designer, you've actually created a class library and imported it into IronPython. This is how easy it is to extend IronPython from other .NET languages, a subject we look into in more detail later in the book.

We've just about made it through our three new commands. You've created commands that can remove pages and rename pages, and written the infrastructure code to support them. In this section, you created a dialog for naming and renaming tab pages, using the forms designer from Visual Studio. In the next section, which introduces a command to add new pages, you get to reuse this dialog.

6.2.4 Adding pages: code reuse in action

When we wrote the dialog to rename pages, we had in mind that you'd also use it to ask the user for a name when creating new pages. If `ShowDialog` is called with `False` as the second argument, then the dialog is shown with an appropriate title for naming a new page.

You already have a method on the tab controller for creating a new tab page: `addTabPage`. This only deals with the view, though; it adds a tab page to the tab control, but it doesn't create a new page on the document. `addTabPage` is called when you load new documents, which already have pages. You need a new method that will handle both the model and the view for you, calling down to `addTabPage` for the view. This is the `newPage` method, shown in listing 6.15. The second argument to `addTabPage` is an empty string because the freshly created page has no text in it yet.

Listing 6.15 Creating a new page in model and adding corresponding tab page

```
def newPage(self, title):
   page = Page(title)
   self.document.pages.append(page)
   self.addTabPage(title, "")
   newPageIndex = len(self.tabControl.TabPages) - 1
   self.tabControl.SelectedIndex = newPageIndex
```

Merely creating a new tab page doesn't select it, so `newPage` finds the index of the new tab page (which is the last one), by asking for the length of the `TabPages` collection.

You have to subtract one from the index because the tab control is zero-indexed, and then you select the new tab page by setting the `SelectedIndex` on the tab control.

With the support for adding tab pages in the tab controller, you need a corresponding command that uses it. This is where the code reuse comes in. The command needs to ask the user for a name, and call `newPage` if the user doesn't cancel out of the dialog. You need to make another call to `ShowDialog`, but pass `False` as the second argument. `NewPageCommand` is shown in listing 6.16.

Listing 6.16 Displaying dialog, checking return value, and creating new page

```
class NewPageCommand(object):
   def __init__(self, tabController):
      self.tabController = tabController

   def execute(self):
      title = ShowDialog("New Page", False)
      if title is not None:
         self.tabController.newPage(title)
```

Default title for a new page

The three tab commands are now complete; but, as usual, they aren't available to the user until they're wired into the view. This small but vital step is the subject of the next section.

6.2.5 Wiring the commands to the view

You've done this before, and these commands are just as easy to wire up as the previous ones. The nice bonus is that adding these commands to the user interface will make MultiDoc actually *usable*. To spice up this section, we add a couple of new features: an application icon and a new document command.

The first is purely cosmetic, but no Windows Forms application would be complete without an application icon to replace the default one! When a form is displayed, an icon is displayed in the upper-left corner of the form, and the same icon is displayed in the taskbar. The default icon is the bizarre three-colored boxes, not unattractive, but also instantly recognizable as the generic Windows Forms icon. Changing this is trivially easy using the `Icon` property on the form and the `Icon` class from the `System.Drawing` namespace.

MultiDoc is a document editor for multiple pages, so an icon showing pages of text is appropriate—and is exactly what the glyFx copy icon looks like. The following code segment should be added to the `MainForm` constructor:

```
iconPath = 'icons\\copy_clipboard_16.ico'
self.Icon = Icon(Path.Combine(executableDirectory, iconPath))
```

The second feature is another that you would consider standard in a document editor: a new document command. With the infrastructure we've provided, this is also easy—but maybe not quite as easy as changing the application icon. Because creating a new document will destroy anything in the current document, you should ask the user for confirmation. (Even better would be to maintain a modified flag so that you

only ask the user for confirmation if the current document has local modifications. We leave this implementation as an exercise for you!)

The `NewDocumentCommand` is almost identical to the `RemoveCommand`—except that, instead of deleting a page, it sets a new document on the tab controller. Listing 6.17 is the `NewDocumentCommand`.

Listing 6.17 `NewDocumentCommand`

```
from System.Windows.Forms import (
   DialogResult, MessageBox,
   MessageBoxButtons, MessageBoxIcon
)

from model import Document

class NewDocumentCommand(object):
   def __init__(self, tabController):
      self.tabController = tabController

   def execute(self):
      result = MessageBox.Show("Are you sure?",
                               "New Document",
                               MessageBoxButtons.OKCancel,
                               MessageBoxIcon.Question)
      if result == DialogResult.OK:
         self.tabController.document = Document()
```

Setting a document on the tab controller triggers the property behavior created earlier, clears the current document, and creates a new one.

The other changes follow the same pattern for wiring the previous commands. The new commands need to be added to `initializeCommands` and then wired up by adding menu and toolbar items. This is just a copy and paste job from the earlier code, slightly modified to use the new commands. The new menu items should appear under a new top-level menu item, Edit.

There's a nice icon for a new document in the glyFx set, but nothing suitable for adding a new page. Justin Fleming of Fuchsia Shock Design[12] came to the rescue and created a plus icon. We haven't shown the menu and toolbar code changes here; they should be easy for you to work out, and the source code is available for download.

With these changes in place, MultiDoc looks like figure 6.8.

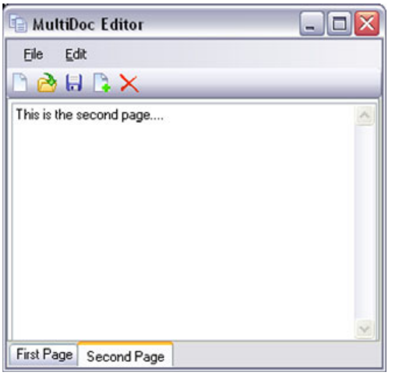


Figure 6.8 The completed MultiDoc with all the new commands added to the user interface

MultiDoc now has all the features from the original specification. To distribute MultiDoc,

[12] http://www.fuchsiashock.co.uk

you currently need to distribute the icons in their original format. This is (normally!) against the glyFx terms of use, so we should find a way of solving this problem.

6.3 Object serializing with BinaryFormatter

One way you can avoid distributing the icons in their original form is to convert them to a convenient binary format. Object persistence, sometimes called serialization, is a common programming need. Serializing takes an object and converts it into binary (or possibly text) data for storing in a file or database. This representation can be turned back into an object again later.

> **Python persistence with pickle**
>
> Python provides standard-library support for object persistence in the form of the `pickle` and `cPickle` modules. Both modules have the same interface.
>
> Under cPython, `cPickle` is a C extension that's faster than `pickle` (a pure Python module). `cPickle` has been implemented for IronPython as a built-in module.
>
> As of IronPython 2, `pickle` and `cPickle` know how to serialize and deserialize .NET objects that support serialization.

You can store the icons as serialized objects; and, instead of constructing `Bitmap` or `Icon` instances inside MultiDoc, you can deserialize them. In the .NET world, the most compact serialization is done with the `BinaryFormatter` class, which lives in the `System.Runtime.Serialization.Formatters.Binary` namespace.

The `BinaryFormatter` is easy to use; you instantiate it and call `formatter.Serialize` with a filestream and the object to persist.

```
>>> import clr
>>> clr.AddReference('System.Drawing')
>>> from System.Drawing import Bitmap
>>> from System.Runtime.Serialization.Formatters.Binary import BinaryFormatter
>>> from System.IO import FileMode, FileStream
>>> bitmap = Bitmap('icons\\save_16.ico')
>>> stream = FileStream('save.dat', FileMode.Create)
>>> formatter = BinaryFormatter()
>>> formatter.Serialize(stream, bitmap)
>>> stream.Close()
```

Re-inflating persisted objects is just as easy.

```
>>> stream = FileStream("save.dat", FileMode.Open)
>>> bitmap = formatter.Deserialize(stream)
>>> stream.Close()
>>> type(bitmap)
<type 'Bitmap'>
```

You can write a simple script that iterates over all the files in the icon directory and converts them into persisted bitmaps (with the exception of the application icon,

which needs to be an `Icon` instead). Python excels at this sort of scripting task; and, with the full range of .NET framework classes available to you, IronPython excels even more. The lack of boilerplate means that you can rapidly write scripts for file manipulation and similar administration jobs.

`Directory.GetFiles` (the equivalent of `os.listdir` from the Python standard library) returns a list of all the files in a directory, so the script shown in listing 6.18 should do the job.

Listing 6.18 Script to serialize all the image files using a `BinaryFormatter`

```
import clr
clr.AddReference('System.Drawing')

from System.Drawing import Bitmap, Icon
from System.IO import Directory, File, Path
from System.Runtime.Serialization.Formatters.Binary import BinaryFormatter

formatter = BinaryFormatter()
iconDirectory = Path.Combine(Directory.GetCurrentDirectory(), 'icons')

exception = 'copy_clipboard_16.ico'
for filePath in Directory.GetFiles(iconDirectory):          # Iterates over the image files
   if not (filePath.endswith('.ico') or filePath.endswith('.gif')):   # Skips non-image files
      continue

   filename = Path.GetFileName(filePath)
   namePart = Path.GetFileNameWithoutExtension(filename)
   outPath = Path.Combine(iconDirectory, namePart + '.dat')

   if filename == exception:
      image = Icon(filePath)
   else:
      image = Bitmap(filePath)

   stream = File.Create(outPath)             # Convenient way to open files
   formatter.Serialize(stream, image)        # Serializes image
   stream.Close()
```

Running this script will save a binary version of all our icons. (Because it skips non-image files, you can run it more than once without it crashing.) You also need to provide a `MainForm` method to deserialize the data files rather than load the images, as shown in listing 6.19.

Listing 6.19 Image deserializing method for `MainForm`

```
def loadImage(self, filename):
   path  = Path.Combine(self.iconPath, filename)
   stream = File.OpenRead(path)
   image = BinaryFormatter().Deserialize(stream)
   stream.Close()
   return image
```

But creating a new `BinaryFormatter` for every image *could* be inefficient. You shouldn't try to optimize *first* unless performance proves to be an issue—premature

optimization is the root of many kinds of evil.[13] Because setting the images is done when MultiDoc loads, you could look at performance tuning if startup time is too long. (And you'd profile how long this delays startup before making the change.) In practice, it doesn't seem to be an issue.

6.4 Summary

We've added five new commands to MultiDoc; now it can be used for the purposes for which it was created: reading, saving, and creating multipage documents. There are other features that might be nice, but none of them should be too difficult to add. The point of the structure we chose for MultiDoc is that it should be relatively easy to work out *where* in the code any changes should go. For example, if you want to save the currently selected tab in the document, you need to add something to the model to represent it. You also need to update the selection on the model when it changes in the view (the job of the tab controller) and so on. Correspondingly, when you're tracking down bugs or problems, you should have a good idea of where to start looking.

We hope that, through MultiDoc, you've seen how easy it is to refactor well-structured Python code to include new features. Particularly useful is the ability to experiment with .NET classes using the interactive interpreter. You can also make minor speculative changes to your code and run it immediately with no need for a compile phase.

Python is no silver bullet—you saw the value of the form designer from Visual Studio—but integrating .NET class libraries with IronPython applications is absurdly easy. Extending your IronPython application with third-party libraries or GUI components is just as simple, or you may want to move performance-sensitive algorithms into C#. IronPython makes it simple to create prototypes, and prove that a particular approach works, before moving code into another .NET language. (In fact, this is how IronPython itself was developed.)

We've now completed MultiDoc as specified. In the process of writing it, I (Michael) discovered several bugs that cost me time to go back and fix. This process would have been a lot less painful if I had a good test suite. One of the advantages of dynamic languages in general, and Python in particular, is that they're easy to test. The ability to modify live objects at runtime comes in particularly useful. In the next chapter, we look at testing practices using the Python standard library module `unittest`.

[13] At least in terms of programming...

7 Agile testing: where dynamic typing shines

This chapter covers

- The Python `unittest` module
- Creating a test framework
- Mock objects
- Testing techniques: monkey patching and dependency injection
- Functional testing

Testing is an increasingly important part of software development. Automated test suites allow you to easily include testing as part of your development process, rather than as a separate, time-consuming, and expensive manual process. The alternative is letting your customers find the bugs for you. Testing isn't only about finding bugs early, though; writing code in a testable way encourages the writing of modular and loosely coupled code—which means better code. Additionally, when adding new features or refactoring, your tests can warn you about what other parts of your code you've broken. Fortunately, testing is an area where dynamic languages particularly shine.

There are many different ways of categorizing tests, and many names for subtly different styles of testing. Broadly speaking, the three categories of tests are as follow:

- Unit tests
- Functional tests[1]
- Regression tests

Unit tests are for testing components of your code, usually individual classes or functions. The elements under test should be testable in isolation from other parts; this isn't always possible, but you have ways of handling these unavoidable dependencies within your tests. Dependencies that need to be managed include cases where your tests need to access external resources like databases or the filesystem.

Functional tests are higher-level tests that drive your application from the outside. They can be done with automation tools, by your test framework, or by providing functional hooks within your application. Functional tests mimic user actions and test that specific input produces the right output. As well as testing the individual units of code that the tests exercise, they also check that all the parts are wired together correctly—something that unit testing alone doesn't achieve.

Regression testing checks that bugs you've fixed don't recur. Regression tests are basically unit tests, but your reason for writing them is different. Once you've identified and fixed a bug, the regression test guarantees that it doesn't come back.

The easiest way to start with testing in IronPython is to use the Python standard library module `unittest`.

7.1 The unittest module

Setting up a test framework with the `unittest` module is easy; simple tests can be set up and run within a matter of minutes. `unittest`, sometimes referred to as `pyunit`, owes its heritage to the Java test framework `JUnit`.[2]

Python testing tools

Testing is a popular pastime among the Python community, and the following tools have approaches to testing that are slightly different from `unittest`:[3]

- nose
- py.test
- doctest
- PyFIT

`doctest` is included in the Python standard library. Unfortunately it doesn't yet work with IronPython because `sys.settrace` isn't implemented. I (Michael) haven't tried the others with IronPython because `unittest` neatly fits the test-first pattern I prefer for development. It may be worth exploring some of these possibilities to see if you prefer them.

---

1 Also known as acceptance, integration, or black-box tests.

2 See http://www.junit.org.

3 For a much more complete reference on a bewildering array of Python testing tools, see http://pycheesecake.org/wiki/PythonTestingToolsTaxonomy.

The basis of using `unittest` is creating test classes, which inherit from `unittest.TestCase`. You pass these to a test runner, which executes all the test methods. Inside the test methods, you create objects, call the functions and methods you're testing, and make assertions about the results.

The test runner runs all your tests and outputs a nicely formatted display of the results. Let's take a closer look at how to use it.

7.1.1 Creating a TestCase

The test runner recognizes any method whose name starts with `test` as a test method. The test runner will call these methods and collect the results.

AssertionError and assert

Unit tests are based on the `assert` statement.

```
assert 2 == 3, "Two is not equal to three"
```

If the expression following the assert statement evaluates to `True`, then execution continues normally. If the expression evaluates to `False`, then an `AssertionError` is raised with the (optional) message supplied following the expression.

When `unittest` runs tests, it catches any `AssertionError`s and marks the test as a failure.

Assertions are made by calling `assert` methods inherited from the `TestCase` class.

You need something to test; and, in order to test it, you need to know what it's supposed to do. Let's create a simple `Value` class for performing operations on numbers. This class should meet the following specifications:

- It should be initialized with a number and store it as a `value` attribute.
- It should have an `add` method, which takes a number and returns the stored value plus the number.
- It should have an `isEven` method, which returns `True` for even numbers and `False` for odd numbers.

From this specification, you can write the tests. Writing the tests first is a process called test-driven development.[4] With appropriately named tests that record the specification (and clearly written tests as well, of course), tests can act as a specification for the code.

Listing 7.1 shows a `TestCase`, called `ValueTest`, which tests the specification just listed.

Listing 7.1 A `TestCase` class for `Value`, which tests the specification

```
import unittest
class ValueTest(unittest.TestCase):
```

[4] Often abbreviated to TDD. See http://en.wikipedia.org/wiki/Test-driven_development.

```
    def testConstructorShouldStoreValue(self):
        value = Value(6)
        self.assertEquals(value.value, 6,
            "value attribute not set correctly")

    def testAddShouldReturnArgumentAddedToValue(self):
        value = Value(6)
        self.assertEquals(value.add(3), 9,
                          "add returned the wrong answer")

    def testIsEvenShouldReturnTrueForEvenNumbers(self):
        value = Value(6)
        self.assertTrue(value.isEven(),
            "Wrong answer for isEven with an even number")

    def testIsEvenShouldReturnFalseForOddNumbers(self):
        value = Value(7)
        self.assertFalse(value.isEven(),
            "Wrong answer for isEven with an odd number")

if __name__ == '__main__':
    unittest.main()
```

The test methods all create an instance of our `Value` class and then test its state or the result of calling methods. The calls to the assert methods do the actual testing, and they all follow a pattern similar to the one in figure 7.1.

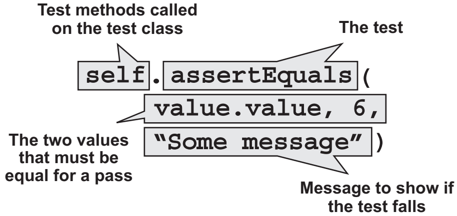


Figure 7.1 The anatomy of an `assert` method on `TestCase`

Several assert methods are available on `TestCase`. In creating a test framework customized for your application, you'll build higher-level tests on these primitive assert methods, or possibly directly with the `assert` statement. Table 7.1 shows the standard assert methods.

Table 7.1 The assert methods available on `TestCase` subclasses. Where relevant, the failure message is always optional.

Method	Usage	Description
`assertEquals`	`self.assertEquals(arg1, arg2, msg=None)`	Asserts that `arg1` and `arg2` are equal.
`assertAlmostEqual`	`self.assertAlmostEqual(arg1, arg2, places=7, msg=None)`	Asserts that `arg1` and `arg2` are almost equal, to the specified number of decimal places. `arg1` and `arg2` should be numeric. Useful for comparing floating point numbers.
`assertTrue`	`self.assertTrue(arg, msg=None)`	Asserts that `arg` evaluates to `True`.

Table 7.1 The assert methods available on `TestCase` subclasses. Where relevant, the failure message is always optional. *(continued)*

Method	Usage	Description
`assertFalse`	`self.assertFalse(arg, msg=None)`	Asserts that `arg` evaluates to `False`.
`assertRaises`	`self.assertRaises(exception, callable, *args)`	Asserts that a specified exception type (the first argument) is raised when `callable` is called. Additional arguments are passed as arguments to `callable`. If no exception, or the wrong exception, is raised, then the test fails.

So what happens when you run the test we've just created? Take a look at figure 7.2.

All the tests error out with a `NameError` because we haven't yet written the `Value` class. This is the TDD approach—to write the tests before the implementation. You'll know when we have a complete implementation; all the tests will pass.

`unittest` formats the results of running the tests in the output. The first line of the output is a line of characters, with one character representing the result of each test.

The three possible results of calling a test method are as follow:



Figure 7.2 The unit tests for `Value`, run without the `Value` class in place

- Everything goes fine, and the test passes. Passing tests are represented by a dot (.).
- The test fails. Failing tests are represented by an 'F'.
- An exception is raised while trying to run the test. Tests that fail because of an unhandled exception are represented by an 'E'.

The last line of the output is the summary of the test run; here all four tests have failed with the same error. Obviously, the missing component is the `Value` class. If we were following strict TDD, you'd implement one method at a time, running the tests in between each method. To save space, listing 7.2 is a full implementation of our enormously complex `Value` class.

Listing 7.2 An implementation of `Value` class, which should pass your tests

```
class Value(object):
   def __init__(self, value):
      self.value = value

   def add(self, number):
      return self.value + number

   def isEven(self):
      return (self.value % 2) == 0
```

When you run the tests now, the output should be much improved (figure 7.3).

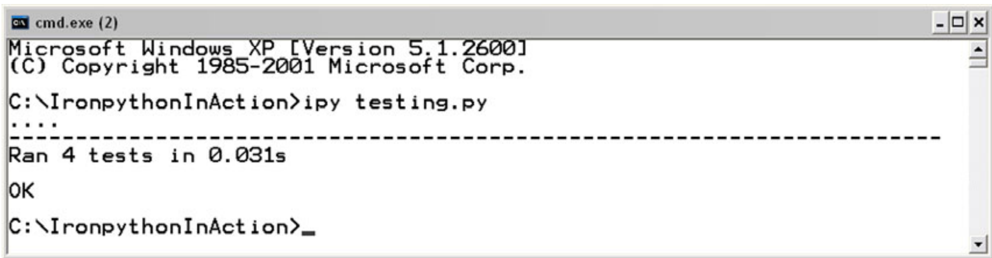


Figure 7.3 The unit tests for `Value`, run with the `Value` class in place

The 'EEEE' from the first run has been replaced with four dots (....), and the summary says OK. Great.

Four out of our five tests start with the same line: `value = Value(6)`. Code duplication is always bad, right? The next section looks at how you can reduce boilerplate in unit tests using `setUp` and `tearDown`.

7.1.2 setUp and tearDown

Some classes can't be tested in isolation; they need either support classes configuring or state initializing before they can be tested. If this initialization includes creating test files or opening external connections, then you may need to both set up tests *and* clean up after them.

`unittest` makes this possible through the `setUp` and `tearDown` methods on `TestCase`. As you might surmise, `setUp` is run before each test method and `tearDown` is run afterward. Any exceptions raised in `setUp` and `tearDown` will count as a test failure.

Listing 7.3 demonstrates how you can use the setUp and tearDown methods with our simple tests.

Listing 7.3 The unit tests for Value rewritten to use setUp and tearDown

```
class ValueTest(unittest.TestCase):
   def setUp(self):
      unittest.TestCase.setUp(self)
      self.value = Value(6)

   def tearDown(self):
      unittest.TestCase.tearDown(self)

   def testConstructorShouldStoreValue(self):
      self.assertEquals(self.value.value, 6,
         "value attribute not set correctly")
...
```

Uses self.value created in setUp

The tearDown shown in this listing isn't strictly necessary, but we wanted to demonstrate how it works. Both setUp and tearDown call up to the base class methods. When creating a test framework for a project, it's common to create a hierarchy of test case classes for different types of tests. Forgetting to call up to the base class setUp is an easy (and sometimes hard to diagnose) mistake to make.

The approach to unittest we've used so far is fine when you have all your tests in a single module. The next section will look at how to create test suites from multiple test files.

7.1.3 Test suites with multiple modules

When writing tests for a project, you'll want to create different test files for different classes and modules in your project. We've been running tests using the unittest.main function. This runs all the tests in whatever is running as the main script. This approach is fine for running each test file separately, but you'll usually want to run all your tests in one go and collect the results.

unittest supports running many tests by allowing you to create test suites from multiple test classes, and passing them to a test runner. (This is what main does under the hood.) Figure 7.4 illustrates the different classes available in unittest to help automate your test running.

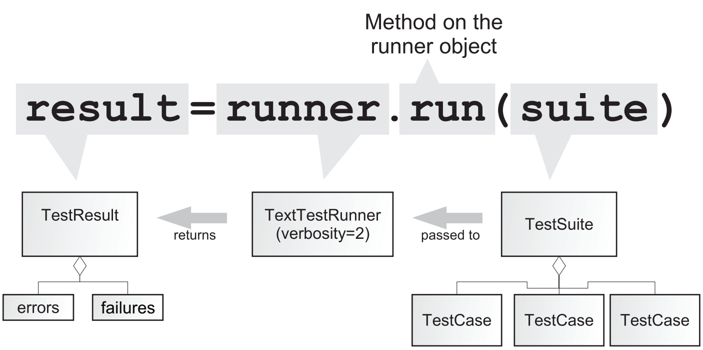


Figure 7.4 Collecting TestCases together in a TestSuite and running them with the TextTestRunner

Using the following components, it's easy to collect our tests and run them all together:

- *TestCase*—The test classes.
- *TestSuite*—Can hold multiple test classes to be executed together.
- *TextTestRunner*—Executes test suites, outputting the results as text. You can control how much information this outputs through the `verbosity` keyword argument.
- *TestResult*—Is returned by the runner and holds information about errors and failures.

To test a large project, you want to be able to collect all the tests together and run them in one go. Using the `__name__ == '__main__'` pattern, it will still be possible to run tests contained in individual modules while developing. Using introspection, you can import the test modules and automatically add all the test classes they contain to a suite. You can then pass the suite to a runner and check the results. Let's create a set of utility functions and keep them in a module called `testutils` (listing 7.4), which we'll be using shortly to test MultiDoc.

Listing 7.4 `testutils`: module to support test running

```
import unittest
from types import ModuleType

def IsTestCase(test):
   if type(test) == type and issubclass(test, unittest.TestCase):      ❶
      return True
   return False

def AddTests(suite, test):
   if isinstance(test, ModuleType):
      for entry in test.__dict__.values():      ❷
         if IsTestCase(entry):
            suite.addTests(unittest.makeSuite(entry))      ❸
   elif IsTestCase(test):
      suite.addTests(unittest.makeSuite(test))      ❸

def MakeSuite(*tests):
   suite = unittest.TestSuite()      ❹
   for test in tests:      ❺
      AddTests(suite, test)
   return suite

def RunTests(suite, **keywargs):      ❻
   return unittest.TextTestRunner(**keywargs).run(suite)
```

The easiest way of explaining these functions is to work from the bottom up.

`RunTests` ❻ runs a test suite and returns the results. It accepts keyword arguments that are passed through to the `TextTestRunner`. `MakeSuite` creates a test suite ❹, and initializes it with any test classes or modules that you pass in ❺. `AddTests` adds tests to a suite ❸. If you pass in a module (recognized because its type is `ModuleType`), then you check every object it contains by iterating over its `__dict__` dictionary ❷, as follows:

```
for entry in test.__dict__.values()
```

Objects are checked to see if they're tests by calling `IsTestCase`, which checks to see if an object is a test class or not ❶. `type(test) == type` is only true for classes. You can tell if a class is a test class by checking if it's a subclass of `TestCase`, as follows:

```
issubclass(test, unittest.TestCase)
```

Listing 7.5 is an example of using these functions to run the unit tests for the `Value` class that you created in the previous sections. It assumes you've put these tests in a module called `ValueTestModule` that you can import into our test runner code.

Listing 7.5 Running tests for `Value` class with help from functions in `testutils`

```
import sys
from testutils import MakeSuite, RunTests

import ValueTestModule        ❶

suite = MakeSuite(ValueTestModule)
results = RunTests(suite, verbosity=2)        ❷

if results.failures or results.errors:
   sys.exit(1)        ❸
```

This listing imports the module containing our tests ❶ and creates the test suite. If you have several test modules, then you can use something like this code segment:

```
import testmodule1
import testmodule2
import testmodule3

suite = MakeSuite(testmodule1, testmodule2, testmodule3)
```

`MakeSuite` adds *all* the modules you pass it to the suite. After creating the suite, you run the tests ❷. Listing 7.5 passes in the optional keyword argument `verbosity=2`. This increases the amount of information output while tests are running—which is useful for keeping track of tests as they run. Figure 7.5 shows the output of running tests with a higher verbosity level.

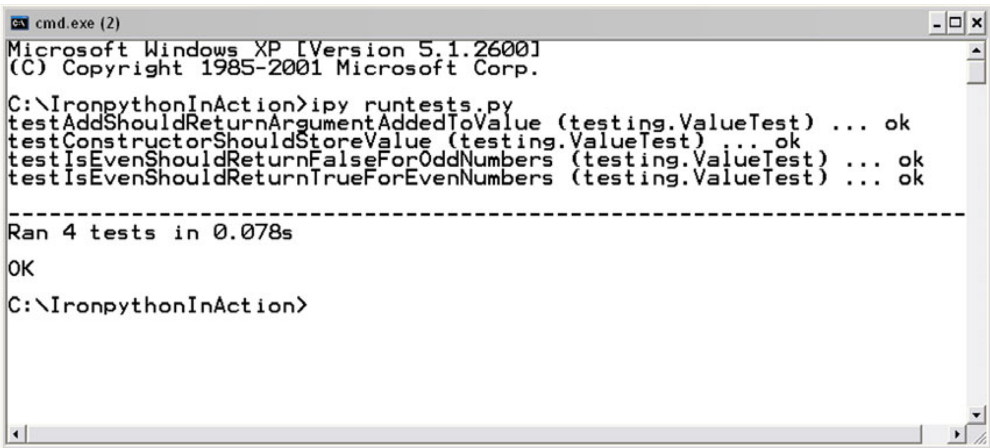


Figure 7.5 Running tests with verbosity level set to 2

Listing 7.5 checks the results of these tests; if there are any errors or failures, it exits with an exit code of 1 ❸. A correct exit code allows you to integrate running your tests into your build process (for example using msbuild)[5] and stop the build if tests fail.

So far, we've concentrated on getting familiar with the basics of the `unittest` framework. This is background to the really interesting part—exploring different testing techniques with IronPython. Many of these techniques make great use of the dynamic nature of Python. To demonstrate this, we write tests for MultiDoc.

7.2 Testing with mocks

The point of unit testing is to test as many aspects of your code as possible in isolation from the other parts; it's easier to tightly specify the behavior of your code and makes the tests run faster. In practice, life is never quite so neat. Your objects need to interact with other objects, and you need to test that interaction. Several different approaches can minimize the number of live objects your tests need to use; the most useful of these approaches is creating mock objects.

7.2.1 Mock objects

Working with mock objects is one place where you'll see the advantage of using a dynamically typed language. In a statically typed language, the compiler won't let you run any code unless the types match those declared in the production code. The objects you use in tests must be real objects, or objects that inherit from the expected type.

In a dynamically typed language, you can take advantage of duck typing. The objects you use for testing need only implement the attributes and methods used in the tests. Let's see this in action by testing the `execute` method of MultiDoc's `OpenCommand`.

> **Test structure**
>
> To make testing easier, we've reorganized the layout of the MultiDoc project. If you download the source code for chapter 7 from the *IronPython in Action* website, then you'll see the changes we've made.
>
> The important changes are that the production modules are kept in a package called `main`, and the test files are in a package called `tests`. A module called `loadassemblies` adds references to all the assemblies you'll use; this can be loaded in the test packages so that you always have references to assemblies you need.

Listing 7.6 gives a quick refresher of what the `execute` code looks like.

Listing 7.6 `OpenCommand.execute`: the method we want to test

```
def execute(self):
   fileName = self.mainForm.document.fileName
   directory = Path.GetDirectoryName(fileName)
   directoryExists = Directory.Exists(directory)
   openFileDialog = self.openFileDialog
```

[5] See http://msdn2.microsoft.com/en-us/library/0k6kkbsd.aspx.

```
if fileName is not None and directoryExists:
   openFileDialog.InitialDirectory = directory
   openFileDialog.FileName = fileName

if openFileDialog.ShowDialog() == DialogResult.OK:
   document = self.getDocument(openFileDialog.FileName)
   if document:
      self.mainForm.document = document
```

The first behavior of `execute` to test is that it correctly sets the `Filename` and `InitialDirectory` attributes on the `OpenFileDialog`. `OpenCommand` has a reference to the `MainForm`, so it can check whether the current document has a `fileName` set on it or not. If this reference isn't `None`, then the filename should be set on the `OpenFileDialog`, so that the dialog opens in the same folder as the current file.

Where possible, *classes should be tested in isolation*. This is one of the ways that testing encourages you to write better code. By making classes easier to test, you end up making them more modular and decoupled from other classes (which is usually in the right direction along the road to *better*).

The only attribute of `MainForm` that `OpenCommand` uses is the `document`. Instead of using a real instance and a real document, you can use a mock object. With a statically typed language, you'd have to provide an object that the compiler recognizes as being of a suitable type. Plenty of libraries could help you to do this, but it's a lot simpler with a dynamic language such as Python. Listing 7.7 shows just about the simplest possible mock object, and how you can use it to create a mock `MainForm` for testing the `OpenCommand`.

Listing 7.7 A simple `Mock` class and a mock `mainform` instance

```
class Mock(object):
   pass
mainform = Mock()
document = Mock()
document.fileName = None
mainform.document = document
command = OpenCommand(mainform)
```

Mocks versus stubs

There's some dispute in the testing community as to whether the kinds of objects we're creating here should be called mocks or stubs.[6]

Those who argue in favor of the name *stubs* use the term *mocks* for a different style of testing. Instead of performing an action and then testing state, which is what we're doing here, they prefer to set up expectations first and then perform the action. If the action doesn't meet the expectations, then the test fails.

Personally, I (Michael) find action followed by assert more natural and easier to read.

[6] For an article on testing by Martin Fowler that strongly argues that these are stubs, see http://martinfowler.com/articles/mocksArentStubs.html.

If you call `execute` as it stands, then the call to `openFileDialog.ShowDialog()` will block—which isn't ideal for an automated test suite. Instead of a real `OpenFileDialog`, we need a mock one with a `ShowDialog` method.

The real dialog box returns either `DialogResult.OK` or `DialogResult.Cancel`. If the user chooses a file, then the returned value is `DialogResult.OK`, which triggers the creation of a new document. In this test, we don't want this result, so we want a class that returns `DialogResult.Cancel` and has `Filename` and `InitialDirectory` attributes (which we *do* want to test). Next, we'll want to test what happens when the user accepts the dialog box. Listing 7.8 shows a `MockDialog` class that allows you to control what's returned from `ShowDialog`.

Listing 7.8 A mock `OpenFileDialog` class

```
class MockDialog(object):
   def __init__(self):
      self.InitialDirectory = None
      self.FileName = None
      self.returnVal = DialogResult.Cancel

   def ShowDialog(self):
      return self.returnVal

command.openFileDialog = MockDialog()
```

Replaces OpenFileDialog with mock

The `OpenCommand` stores the dialog as an instance attribute, so replacing it with the mock dialog is easy.

Now to use this in a test. In listing 7.9, the test name reflects the behavior we're testing. Because you'll need an `OpenCommand` initialized with a mock `MainForm` and dialog in the next few tests, we place this code in a `setUp` method.

Listing 7.9 Testing the use of `OpenFileDialog`

```
def setUp(self):
   mainform = Mock()
   document = Mock()
   document.fileName = None
   mainform.document = document

   self.command = OpenCommand(mainform)
   self.command.openFileDialog = MockDialog()

def testExecuteShouldSetFilenameAndInitialDirectoryOnDialog(self):
   self.command.execute()
   self.assertNone(self.command.openFileDialog.FileName,
             "FileName incorrectly set")

   self.command.mainForm.document.fileName = __file__
   self.command.execute()
   self.assertEquals(self.command.openFileDialog.FileName,
              __file__,
              "FileName incorrectly set")
   self.assertEquals(self.command.openFileDialog.InitialDirectory,
              Path.GetDirectoryName(__file__),
              "InitialDirectory incorrectly set")
```

Tests behavior with no filename

Sets a real filename

The mock objects we've created in this section are pretty specific to testing the `OpenCommand`. Instead of creating all your mocks from scratch, you *could* use any of the following Python mock libraries:

- *Mock*—http://pypi.python.org/pypi/mock/
- *Mox*—http://code.google.com/p/pymox
- *The Python Mock Module*—http://python-mock.sourceforge.net/
- *pMock*—http://pmock.sourceforge.net/
- *pymock*—http://pypi.python.org/pypi/pymock/
- *pymockobject*—http://pypi.python.org/pypi/pymockobject/
- *minimock*—http://blog.ianbicking.org/minimock.html
- *Stubble*—http://www.reahl.org/project?name=stubble

Perhaps the reason for the many different mock libraries is that they're so easy to write. The first one in the list is my favorite, because I wrote it, but it's often simpler to write mocks as you need them as we've been doing.

Even though creating mocks is easy, that doesn't mean that there aren't reusable patterns. In the next section, we look at a slightly different testing pattern using a `Listener` class.

7.2.2 Modifying live objects: the art of the monkey patch

Next, we need to test the behavior of `execute` when the dialog is accepted or canceled. If the dialog is accepted, it calls the `getDocument` method with the filename from the dialog and sets the returned document onto the `MainForm`. If the dialog is canceled, then it doesn't.

You could test this by providing a known filename and then checking that the returned document is valid and complete. The test would need a real file, and would also depend on the `Document` class remaining the same. If you changed `Document`, then you'd also need to change the way you test `OpenCommand`. This is why testing in isolation is preferred—tests for one part of the code become much less brittle to changes in another part of the code.

You can get around this by replacing the `getDocument` method with a custom object that returns a mock document and also allows you to confirm whether it has been called or not.

Adding methods at runtime is another feature of dynamic languages, but is known among the Python community by the slightly pejorative term of *monkey patching*. The reason it's frowned on is that it can make your code hard to read. If a class defines a method, and then later on you see that method being called, you'll assume you know what code is being executed. If in fact that method has been replaced, it's difficult to know what code is being executed.

One place where monkey patching is both accepted and useful is in testing. Because methods are looked up dynamically, you can add methods at runtime. In order to understand monkey patching, it will be useful to take a brief look at the Python attribute lookup rules.

NOTE Monkey patching is a term that started with the Python community but is now widely used (especially within the Ruby community). It seems to have originated with Zope[7] programmers, who referred to *guerilla patching*. This evolved from gorilla patching into monkey patching.

ATTRIBUTE LOOKUP RULES

When you call a method on an instance, the method is looked up using the normal order shown in figure 7.6.[8]

You can confirm these rules at the interactive interpreter by adding a new method to a class. All instances of the class then gain the method.

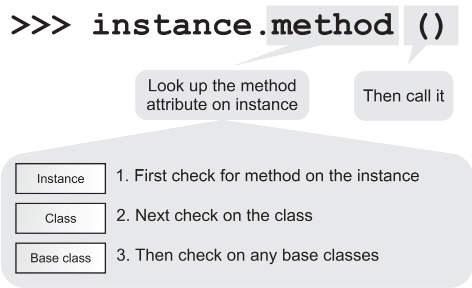


Figure 7.6 How a method call becomes an attribute lookup followed by a call

```
>>> class AClass(object):
...     pass
...
>>> instance = AClass()
>>> def method(self):
...     print 'Hello'
...
>>> AClass.method = method
>>> instance.method()
Hello
```

In our case, you have the choice of patching the replacement object on the class or on the instance. The disadvantage of patching the class is that it's effectively a global and modifying it will also modify it for other tests. You can override methods used at runtime by patching the instance—which is what we need for testing. Again, this is easy to show in an interactive interpreter session.

```
>>> def method2():
...     print 'Hello 2'
...
>>> instance.method = method2
>>> instance.method()
Hello 2
```

We've now covered most of the basic principles of testing in Python, useful knowledge to apply whether you're programming in CPython or IronPython. Coming soon is functional testing, but first we put monkey patching into practice with a useful test class.

[7] Zope is a large Python web framework that was first released in 1998. It's mainly used to create Content Management Systems (CMS). The most famous application built with Zope is a popular CMS called Plone.

[8] Because of the descriptor protocol, the lookup order and rules are a bit more complex—but figure 7.6 shows the basic principle.

Bound and unbound methods

You may have noticed that when you patched the class with a function, it needed to take the argument `self` like normal methods. When you look up a method defined on a class, it gets wrapped as a bound method object—which is how `self` is passed in. (This happens because functions are descriptors.) When you attach a function to an instance, it's just an ordinary attribute and so doesn't get `self` passed to it.

Bound methods are named because of the way `self` is bound to the method.

This distinction between bound methods (methods looked up on an instance and wrapped as a bound method) and unbound methods (the same object fetched directly from the class) isn't only deep Python; it can also be very useful.

.NET has a similar distinction with closed static and open instance delegates. Good references are hard to come by, but the best we've found[9] is at http://peisker.net/dotnet/languages2005.htm.

THE LISTENER CLASS

We want to test the behavior of `OpenCommand.execute`. If the call to `ShowDialog` returns `DialogResult.OK`, then `getDocument` should be called with the filename from the dialog, and the return value should be set as the document on `MainForm`. You need to monkey-patch `getDocument` with something that will record what arguments it's called with and lets you control what it returns.

You could write a function that will record these things, but a useful pattern provides a general solution to this need: the `Listener` class (listing 7.10).

Listing 7.10 `Listener` class that records arguments it's called with

```
class Listener(object):
   def __init__(self):
      self.reset()

   def reset(self):
      self.returnVal = None
      self.triggered = False
      self.triggerArgs = None
      self.triggerKeyWargs = None

   def __call__(self, *args, **keywargs):
      self.triggered = True
      self.triggerArgs = args
      self.triggerKeyWargs = keywargs
      return self.returnVal
```

The magic of the `Listener` is in the `__call__` method, which is another one of Python's magic methods. Instances of objects that define a `__call__` method are callable like functions; but, because they're class instances, they can store state.

[9] Actually, thanks to Seo Sanghyeon, who found this for us.

This `__call__` method collects all the arguments it's called with (using `*args` and `**keywargs`) and stores them on the instance. When you monkey-patch a method with a `Listener` instance you can tell whether it has been called and with what arguments. Let's use this to test the behavior of `execute` when the dialog is accepted or canceled. The first part of this test, shown in listing 7.11, is easy. You assert that, when the dialog is canceled, `getDocument` *isn't* called.

Listing 7.11 Testing that `getDocument` isn't called if the dialog is canceled

```
def testExecuteShouldNotCallGetDocumentForCancelledDialog(self):
   listener = Listener()
   self.command.getDocument = listener

   self.command.execute()
   self.assertFalse(listener.triggered, "getDocument called incorrectly")
```

Listing 7.11 is trivially simple; you're just testing that `getDocument` *isn't* called. Its more important counterpart test is listing 7.12.

Listing 7.12 Testing that accepting the dialog *should* call `getDocument`

```
def testExecuteWithAcceptedDialogShouldCallGetDocument(self):
   listener = Listener()
   self.command.getDocument = listener

   originalDocument = self.command.mainForm.document
   self.command.mainForm.document.fileName = __file__
   self.command.openFileDialog.returnVal = DialogResult.OK
   self.command.execute()
   self.assertEquals(listener.triggerArgs, (__file__,),
                "getDocument not called with filename")
   self.assertEquals(self.command.openFileDialog.InitialDirectory,
                Path.GetDirectoryName(__file__),
                "FileName incorrectly set")
   self.assertEquals(self.command.mainForm.document,
                originalDocument,
                "document incorrectly changed")
```

Here you're testing what happens if the dialog is accepted, but `getDocument` returns `None` (the default). If the dialog is accepted, then `getDocument` should be called with the filename set on the dialog. Because `getDocument` returns `None`, the document on `MainForm` should *not* be replaced.

The last part of this test is listing 7.13, which tests that the document *is* replaced when it should be.

Listing 7.13 Testing `getDocument` and `MainForm` interaction

```
def testNewDocumentFromGetDocumentShouldBeSetOnMainForm(self):
   listener = Listener()
   self.command.getDocument = listener
   self.command.mainForm.document.fileName = __file__
   self.command.openFileDialog.returnVal = DialogResult.OK
```

```
newDocument = object()
listener.returnVal = newDocument
self.command.execute()
self.assertEquals(self.command.mainForm.document,
             newDocument,
             "document not replaced")
```

Although the `Listener` is only a small class, it opens up the way to an effective and readable testing pattern.

But there's a potential problem with monkey patching. Your tests become white-box tests that know a great deal about the implementation of the objects under test. For example, if you change the implementation of `getDocument` so that it takes two arguments instead of one, the tests we've written so far would continue to pass even though the code is broken. Finding this balance between testing implementation and testing behavior is a constant tension in unit testing. One pattern that can help reduce this coupling is dependency injection.

7.2.3 Mocks and dependency injection

In dependency injection, dependencies are supplied to components rather than being used directly. Dependency injection makes testing easier, because you can supply mocks instead of the real dependencies and test that they're used as expected. A common way to do dependency injection in Python is to provide dependencies as default arguments in object constructors.[10]

Let's look at testing a simple scheduler class to see how this works (listing 7.14).

Listing 7.14 A simple `Scheduler` class to test with dependency injection

```
import time

class Scheduler(object):
   def __init__(self, tm=time.time, sl=time.sleep):
      self.time = tm
      self.sleep = sl

   def schedule(self, when, function):
      self.sleep(when - self.time())
      return function()
```

`Scheduler` has a single method, `schedule`, that takes a callable and a time for the callable to be fired. The `schedule` method blocks by sleeping until the correct time (`when`) using the `time.time` and `time.sleep` standard library functions; but, because it obtains them with dependency injection, it's easy to test. The injection is set up in the `Scheduler` constructor, so the first thing you need to test is that the default constructor does the right thing. Setting up the default dependency in the constructor is the extra layer that dependency injection introduces into your code. Listing 7.15 shows the test for the constructor.

[10] With thanks to Alex Martelli. Examples adapted from http://www.aleax.it/yt_pydi.pdf.

Listing 7.15 Testing that dependency injection is set up correctly

```
import time
from unittest import TestCase
from dependency_injection import Scheduler

class DependencyInjectionTest(TestCase):

   def testConstructor(self):
      scheduler = Scheduler()
      self.assertEquals(scheduler.time, time.time,
                   "time not initialized correctly")
      self.assertEquals(scheduler.sleep, time.sleep,
                   "sleep not initialized correctly")
```

Having tested that the dependency injection is properly initialized in the default case, you can use it to test the `schedule` method. Listing 7.16 uses a fake `time` module that records calls to `time`, `sleep`, and the function you pass into `schedule`. Methods on `FakeTime` are passed into the constructor instead of the defaults. You can then assert that calls are made in the right order, with the right arguments, and that `schedule` returns the right result.

Listing 7.16 Testing `schedule` method by injecting faked-up dependencies

```
def testSchedule(self):
   class FakeTime(object):
      calls = []

      def time(self):
         self.calls.append('time')
         return 100

      def sleep(self, howLong):
         self.calls.append(('sleep', howLong))

   faketime = FakeTime()
   scheduler = Scheduler(faketime.time, faketime.sleep)

   expectedResult = object()
   def function():
      faketime.calls.append('function')
      return expectedResult

   actualResult = scheduler.schedule(300, function)

   self.assertEquals(actualResult, expectedResult,
     "schedule did not return result of calling function")

   self.assertEquals(faketime.calls,
         ['time', ('sleep', 200), 'function'],
         "time module and functions called incorrectly")
```

Because the fake `time` function is set up to return 100, and the function is scheduled to be called at 300, `sleep` should be called with 200. Dependency injection can easily be done using setter properties, or even with simple attributes. Yet another approach is to use factory methods or functions for providing dependencies, which can be needed where fresh instances of dependencies are required for each use. Dependency injection

is useful for both unit testing and subclassing, or overriding the behavior of classes; subclassing is significantly easier to do with Python than some other languages.

One of the problems with unit testing is that, although it's good for testing components in isolation, it isn't so good at testing that they're wired together correctly. To make sure that this is covered, you need some higher-level tests; this is where functional testing comes in.

7.3 Functional testing

Functional tests, or acceptance tests, are high-level tests of an application from the outside. As much as possible, they should interact with the application in the same way the user does. Where unit tests test the components of your application, functional tests test the interaction of those components; they'll often pick up on bugs or problems that unit tests miss. Functional tests can be more than just useful tests, though.

In the Extreme Programming (XP) tradition (you know a methodology has arrived when it becomes a tradition), new features are specified by the customer as user stories.[11] User stories describe (and specify) the behavior of your application. A functional test then becomes an executable version of this user story. If you follow XP, then your user stories provide a full specification of your application's behavior. As well as testing your components, functional tests will warn you when new features interact in unexpected ways with existing features.

In the first part of this section, we write a functional test to test the New Tab Page feature, so we need a user story. The user story describes a feature from the point of view of a user, and our user will be called Harold.[12]

1. Harold opens MultiDoc.
2. He clicks the New Page toolbar button.
3. A dialog called Name Tab appears.
4. Harold changes his mind, so he selects Cancel.
5. No new tab appears.
6. Our capricious user clicks the button again.
7. The dialog appears again.
8. This time he enters a name: My New Page.
9. He clicks OK.
10. There are now two tabs.
11. The second one is called My New Page.
12. Harold is ecstatic.

This user story specifies how the New Tab Page dialog should work, and how the user interacts with it. We need to implement a test that follows Harold's actions and checks that MultiDoc behaves in the expected way.

[11] See http://www.extremeprogramming.org/rules/userstories.html.

[12] This is in homage to the long-suffering, but perhaps ever so slightly demented, star of the Resolver Systems user stories.

This is where it gets tricky. We want to use our existing test framework, so that our functional tests can be first-class members of the automated test suite, but you need a way of driving and interacting with MultiDoc. You need to create a new test case class that provides the infrastructure for writing functional tests. This test case should start MultiDoc for you and allow you to interact with it. This in turn gives us a new problem to solve: how do you interact with a running MultiDoc?

7.3.1 Interacting with the GUI thread

You need to start MultiDoc, perform actions, and then make assertions about the state of MultiDoc. *But*, starting MultiDoc means starting the Windows Forms event loop, which will seize the control flow of the thread. The logical thing to do is start MultiDoc on another thread. As if we didn't have enough problems already, doing this will create another one—any interaction with Windows Forms controls has to be on the thread on which they were created. Fortunately, this is all relatively simple to do. You need to know the following three facts:

- The Windows Forms event loop must run in a Single Threaded Apartment (STA) thread.[13]
- Windows Forms controls provide an `Invoke` method, which takes a delegate and executes on the control thread. It's synchronous, so `Invoke` can return values but blocks until execution has completed.
- IronPython provides a convenient delegate that you can create with a function and use with Invoke. This delegate is called `CallTarget0`, and where you import it from depends on which version of IronPython you're using.

Listing 7.17 shows a simple example of starting the event loop on another thread.

Listing 7.17 Interacting with Windows Forms controls from another thread

```
import clr
clr.AddReference('System.Windows.Forms')
from System.Windows.Forms import Application, Form
from System.Threading import (
   ApartmentState,
   Thread, ThreadStart
)

try:
   # IronPython 1.x
   from IronPython.Runtime.Calls import CallTarget0
except ImportError:
   # IronPython 2.x
   clr.AddReference('IronPython')
   from IronPython.Compiler import CallTarget0

class Something(object):     <-- Somewhere to store attributes
   started = False
```

[13] Because they wrap native controls that assume they'll run in a thread with an STA state.

```
    form = None

something = Something()

def StartEventLoop():
    f = Form()
    f.Text = 'A Windows Forms Form'
    f.Show()
    something.form = f          <-- Stores reference to form
    something.started = True
    Application.Run(f)

thread = Thread(ThreadStart(StartEventLoop))
thread.SetApartmentState(ApartmentState.STA)
thread.Start()

# Some time for form to appear
while not something.started:          <-- Waits for form to be shown
    Thread.CurrentThread.Join(100)

def GetFormTitle():
    title = something.form.Text
    return title

title = something.form.Invoke(CallTarget0(GetFormTitle))
print title
                                    A brief pause to see form
Thread.Sleep(5000)        <--
delegate = CallTarget0(something.form.Close)
something.form.Invoke(delegate)
```

The delegate `CallTarget0` wraps functions that don't take any arguments. A corresponding `CallTarget1` wraps functions taking one argument, `CallTarget2` for functions that take two arguments, and so on up to `CallTarget5`.[14] We find it simpler to use `CallTarget0` and pass in lambda functions where we need a function called with multiple arguments.

Listing 7.18 puts this knowledge to work with a new base class for tests: `FunctionalTest`. The `setUp` method starts MultiDoc and `tearDown` stops it. `FunctionalTest` also provides a convenience method `invokeOnGUIThread` for interacting with MultiDoc by executing functions on the GUI thread.

Listing 7.18 A `FunctionalTest` base class for interacting with a running MultiDoc

```
from tests.testcase import TestCase

from main.mainform import MainForm

from System.IO import Directory, Path
from System.Windows.Forms import Application
from System.Threading import (
    ApartmentState,
    Thread, ThreadStart
)
```

[14] .NET 3 also has two useful delegates with various aritys. These are `Action` and `Func`.

```
try:
   from IronPython.Runtime.Calls import CallTarget0
except ImportError:
   from IronPython.Compiler import CallTarget0

class FunctionalTest(TestCase):

   def setUp(self):
      self.mainForm = None
      self._thread = Thread(ThreadStart(self.startMultiDoc))
      self._thread.SetApartmentState(ApartmentState.STA)
      self._thread.Start()
      while self.mainForm is None:
         Thread.CurrentThread.Join(100)

   def startMultiDoc(self):
      fileDir = Path.GetDirectoryName(__file__)
      executableDir = Directory.GetParent(fileDir).FullName

      self.mainForm = MainForm(executableDir)
      Application.Run(self.mainForm)

   def tearDown(self):
      self.invokeOnGUIThread(lambda: self.mainForm.Close())

   def invokeOnGUIThread(self, function):
      return self.mainForm.Invoke(CallTarget0(function))
```

Imports CallTarget0 from IronPython 1

Imports CallTarget0 from IronPython 2

New subclass of TestCase

This new test case is nice, and it will work fine (trust me; we've already tried it). But it isn't quite sufficient for what we want to achieve. We want to test the New Page dialog; and, if you invoke a function on the control thread that opens the dialog, `Invoke` will block until the dialog is closed again. You need a way to asynchronously perform actions on the control so that you can interact with the Name Tab dialog. This means more fun with dialogs.

7.3.2 An AsyncExecutor for asynchronous interactions

You can use a similar pattern to interact asynchronously with the GUI thread. You can launch the action from yet another thread that has the job of calling `Invoke`. This won't block the test thread while it's waiting for `Invoke` to return. You may want to be able to retrieve a return value, and to be able to join to the new thread to check that it exits, you can encapsulate this functionality in an object. Listing 7.19 shows the `AsyncExecutor` object along with a convenience method to use it from functional tests.

Listing 7.19 A `FunctionalTest` base class for interacting with a running MultiDoc

```
from System.Threading import (
   ManualResetEvent, Timeout
)
class AsyncExecutor(object):

   def __init__(self, function):
      self.result = None
      startEvent = ManualResetEvent(False)

      def StartFunction():
```

```
            startEvent.Set()
            self.result = function()

        self._thread = Thread(ThreadStart(StartFunction))
        self._thread.Start()
        startEvent.WaitOne()                               <- Waits for ManualResetEvent to signal

    def join(self, timeout=Timeout.Infinite):
        return self._thread.Join(timeout)      <- Joins execution thread

---

    def executeAsynchronously(self, function):             <- Convenience method for FunctionalTest
        def AsyncFunction():
            return self.invokeOnGUIThread(function)
        executor = AsyncExecutor(AsyncFunction)
        return executor
```

Now you have all the infrastructure you need to write the functional test. Ideally, the test would know nothing about the internal structure of MultiDoc; but in order to make assertions about the state of MultiDoc, it needs to know *something*. This makes the test brittle against changes to the structure of MultiDoc. In the next section, we turn our user story into a functional test while looking at how you can mitigate against this potential brittleness.

7.3.3 The functional test: making MultiDoc dance

You need to create a test file and add it to runtests.py. The new test inherits from the test case, `FunctionalTest`. `setUp` automatically launches MultiDoc and gives you access to the main class (the `MainForm` instance) as `self.mainForm`. We still need to work out how you'll interact with MultiDoc.

One way would be to insert fake mouse and key events into the event loop. A managed class is available for sending key presses to the Windows Forms message loop. There are no managed classes for sending mouse movements and button presses, but you can do this using unmanaged classes.[15] Even taking this route, you'd still need access to the controls to get the locations to send clicks to. If you're going to have access to the controls anyway, then you might as well trigger them programmatically. This approach still tests that event handlers are wired correctly, and is a good compromise between fully black-box testing and testing completely below the level of the GUI. The advantage of simulating mouse moves and clicks is that it only works if your GUI components are accessible to the mouse (that is, visible). The cost is having to maintain a more complex test framework.

The first important step in our user story is clicking the New Page toolbar button. This button is the fourth button in the toolbar, so you can access it using code like `mainForm.toolBar.Items[3].PerformClick()` (which must be executed on the control thread, of course). This suffers from the brittleness we mentioned earlier. If you change the order of buttons in the toolbar, then you have to modify everywhere that uses this code. A simple solution is to access the button through a single method. This

[15] This is the approach Resolver Systems takes with the test framework for Resolver One.

has all the usual advantages of avoiding duplication and means that you can change the way you access toolbar buttons from a single place. Listing 7.20 shows the start of the functional test and a method for clicking the New Page button.

Listing 7.20 A `FunctionalTest` base class for interacting with a running MultiDoc

```
from tests.functionaltest import FunctionalTest

from System.Threading import Thread
from System.Windows.Forms import Form, SendKeys

class NewPageTest(FunctionalTest):

    def clickNewPageButton(self):
        button = self.mainForm.toolBar.Items[3]
        executor = self.executeAsynchronously(
                    lambda: button.PerformClick())
        Thread.CurrentThread.Join(200)
        return executor
```

Delay for dialog to appear (annotation on `Thread.CurrentThread.Join(200)`)

Putting the access to UI components into methods also has the advantage of making the functional test more readable. As you write more functional tests, and abstract out of them methods for common actions, you effectively create a Domain Specific Language (DSL) for writing your tests. The ideal would be to have one line of code per line of user story.

An alternative way of making tests less susceptible to breakage caused by layout changes is to give controls a descriptive `Name` attribute. It's then easy to provide a `findControlByName` method that recursively iterates through child controls looking for a specific control. You don't need to store a reference to all the controls you might want to access through functional tests, and changing your layout won't (necessarily) break all your functional tests.

Executing our functional test will cause MultiDoc to appear along with dialog and new tab pages. Following the user story, MultiDoc will dance under the invisible hand of Harold. (Because this is automated, the actual dance is very quick, but it will look like figure 7.7.)

The full functional test is shown in listing 7.21. It contains the user story as comments in the test method, each line followed by the code that implements it.

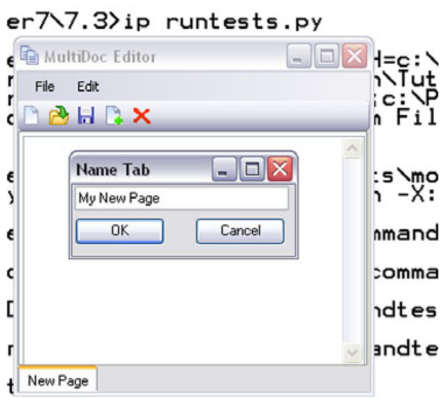


Figure 7.7 MultiDoc dancing under the invisible hand of Harold, our mythical user

Listing 7.21 `FunctionalTest` base class for interacting with running MultiDoc

```
def testNewPageDialog(self):
    # * Harold opens MultiDoc
    # * He clicks on the 'New Page' toolbar button
```

Done by setUp (annotation on `# * Harold opens MultiDoc`)

```
executor = self.clickNewPageButton()

# * A dialog called 'Name Tab' appears
dialog = self.invokeOnGUIThread(
          lambda: Form.ActiveForm)                    <-- Dialog should be active form
title = self.invokeOnGUIThread(lambda: dialog.Text)
self.assertEquals(title, "Name Tab",
             "Incorrect dialog name")

# * Harold changes his mind, so he selects cancel
self.invokeOnGUIThread(
   lambda: dialog.CancelButton.PerformClick())
executor.join()

# * No new tab appears
def GetNumberOfPages():
   return len(self.mainForm.tabControl.TabPages)
numPages = self.invokeOnGUIThread(GetNumberOfPages)
self.assertEquals(numPages, 1,
             "Wrong number of tabPages")

# * Our capricious user clicks the button again
executor = self.clickNewPageButton()

# * The dialog appears again
dialog = self.invokeOnGUIThread(lambda: Form.ActiveForm)

# * This time he enters a name 'My New Page'
def TypeName():
   SendKeys.SendWait('My New Page')
self.invokeOnGUIThread(TypeName)

# * He clicks on OK
self.invokeOnGUIThread(
   dialog.AcceptButton.PerformClick())

# * There are now two tabs
numPages = self.invokeOnGUIThread(GetNumberOfPages)
self.assertEquals(numPages, 2,
             "Wrong number of tabPages")

# * The second one is called 'My New Page'
def GetSecondTabTitle():
   secondTab = self.mainForm.tabControl.TabPages[1]
   return secondTab.Text

secondTabTitle = self.invokeOnGUIThread(GetSecondTabTitle)
self.assertEquals(secondTabTitle, "My New Page",
             "Wrong title on new page")

# * Harold is ecstatic    <-- Not sure what assert to use
```

As you can see, several pieces of code in the test still poke inside MultiDoc to test its state. If you were to implement more tests, you'd find that a lot of this code could be shared between tests and moved up to become methods on `FunctionalTest`. In this way, the tests become more DSL-like, and you build a comprehensive test framework.

That was fun, but it might give you the impression that creating a functional test suite for your application is easy. We didn't have to deal with timing or threading

issues at all. Despite potential difficulties, getting functional tests working is the most satisfying part of writing tests. We've now finished with testing, so it's time to wrap up.

7.4 Summary

In this chapter, we've gone from setting up a `unittest`-based test framework to the principles of testing with Python. The dynamic nature of Python makes it easy to test.

It seems obvious that testing is an important part of software development. A good principle is that if you have untested code, you can't be sure it works. As you add new features, good test coverage tells you if you've broken any of your existing features—an enormous benefit. A less obvious benefit comes when refactoring code. Changing classes that you use extensively, to extend or improve the architecture of your code, can be a daunting task. Your initial changes may break a lot, but with good test coverage you'll know you've finished when all your tests pass again!

There's a lot we haven't covered in this chapter, but the basic examples of monkey patching and mocks that we've used can be extended to provide solutions to difficult testing situations. Because dynamic languages are so easy to test, lots of people are exploring testing with Python, and lots of resources on the internet are available to help you.

Along the way, you encountered another of Python's magic methods, `__call__`, for creating callable objects. This method provides access to the Python equivalent of .NET interfaces, plus things that aren't possible with C# or VB.NET. In the next chapter, we explore some more of these Python protocols. Not everything in the .NET framework maps to Python syntax or concepts straightforwardly. In the next chapter, we also look at some of the ways that IronPython integrates with the .NET framework. These are things that past experience with Python or .NET alone hasn't equipped you for.

8 Metaprogramming, protocols, and more

This chapter covers

- Python protocols
- Dynamic attribute access
- Metaprogramming with metaclasses
- Advanced .NET interoperation

In this chapter, we're going to look under the hood of the Python programming language. We've covered all the basic syntax and how to use classes from the .NET framework with IronPython. To make full use of Python, you need to know how to hook Python classes into the infrastructure that the language provides. As you write classes and libraries in Python, you'll need to define how your objects take part in normal operations and interact with other objects. Much of this interaction is done through protocols, the magic methods that we've already briefly discussed.

We focus here on the use of protocols, including the metaprogramming capabilities of Python in the form of metaclasses. Metaclasses have a reputation for being something of a black art, but no book on Python would be complete without them. They can be used to achieve tasks that are much harder or even impossible

with other approaches. In this part of the chapter, you'll deepen your understanding of Python as you learn about the most important protocols.

The next part of the chapter looks at how IronPython integrates with .NET. Most of the time IronPython sits easily with the way .NET does things, but sometimes they don't make such comfortable bedfellows. In these situations, you need to know how .NET functionality is made available in IronPython. This information will prove invaluable in any non-trivial project; along the way, we take a closer look at some of the .NET types that you use through IronPython.

Let's start by examining how Python protocols relate to interfaces, a topic that .NET programmers will already be familiar with.

8.1 Protocols instead of interfaces

Interfaces are used in C# to specify behavior of objects. For example, if a class implements the `IDisposable` interface, then you can provide a `Dispose` method to release resources used by your objects. .NET has a whole host of interfaces, and you can create new ones. If a class implements an interface, it provides a static target for the compiler to call whenever an operation provided by that interface is used in your code.

NOTE C# does have one example of duck typing: enumeration with the `IEnumerable` interface. To support iteration over an object,[1] classes can *either* implement `IEnumerable` *or* provide a `GetEnumerator` method. `GetEnumerator` can either return a type that implements `IEnumerator` or one that declares all the methods defined in `IEnumerator`.

In Python, you don't need to provide static targets for the compiler, and you can use the principle of duck typing. Many operations are provided through a kind-of-soft interface mechanism called protocols. This isn't to say that formal interface specification is decried in Python—how could you use an API if you didn't know what interface it exposed?—but, again, Python chooses not to *enforce* this in the language design.

NOTE Various third-party interface packages are available for Python, the most common one being part of the Zope project.[2]

In this section, we look at the common Python protocols and how to implement them.

8.1.1 A myriad of magic methods

Methods that implement protocols in Python usually have names that start and end with double underscores. They're known as the magic methods because, instead of you calling them directly, they're usually called *for you* by the interpreter.

When you compare objects, the comparison operation will cause the method corresponding to the operation to be called magically (as long as the object provides the appropriate protocol method).

We've already encountered several of the protocols in our journey so far: the `__init__` constructor and the `__getitem__` and `__setitem__` methods that implement

[1] From http://msdn2.microsoft.com/en-us/library/ttw7t8t6(VS.71).aspx.
[2] The `zope.interface` package: http://www.zope.org/Products/ZopeInterface.

the mapping and sequence protocols. As you'd expect, there are many more that we haven't yet encountered. A full list of all the common magic methods is in one of the appendixes, but some of them are so central to Python that we can't do justice to the language without looking at them.

As the heading implies, Python has a lot of magic methods. You implement these to customize the behavior of your objects. Some of the protocols central to Python are also simple to implement. We start with one of these, the `__len__` protocol method.

THE LENGTH OF CONTAINERS

In .NET land, you find the number of members in a container object through the `Length` property or the `Count` property, depending on what kind of container it is. In Python, you determine the length of a container by calling `len` on it. Under the hood, this calls the `__len__` method (listing 8.1).

Listing 8.1 Specifying length of custom containers

```
>>> class Container(object):
...     def __len__(self):
...         return 3
...
>>> container = Container()
>>> len(container)
3
>>>
```

This listing illustrates a silly implementation of `__len__` that always returns 3. The body of this method is likely to contain slightly more logic in a real container class.

That was easy. For our next trick, we look at the Python equivalents of `ToString`.

REPRESENTING OBJECTS AS STRINGS

You didn't read that last sentence wrong—it's meant to be plural. You can explicitly get a string version of an object by calling `str` on it. `str` is for producing a pretty version of an object. You can also get a representation of an object with another built-in function called `repr`.

You can see this distinction by working at the interactive interpreter with strings containing escaped quotes.

```
>>> x = 'string with \'escaped\' quotes'
>>> str(x)
"string with 'escaped' quotes"
>>> repr(x)
'"string with \'escaped\' quotes"'
>>>
```

The stringified version doesn't have the escape backslashes. The `repr`'d version does have them; and, because the `repr`'d version includes quotes, the console adds an extra set.

When you call `str` on an object, Python calls `__str__` for you. When you use `repr`, Python calls `__repr__`. Naturally these methods must return strings, as shown in listing 8.2.

Listing 8.2 A class with custom string representations

```
>>> class Something(object):
...     def __str__(self):
...         return 'Something stringified'
...     def __repr__(self):
...         return "Something repr'd"
...
>>> something = Something()
>>> str(something)
'Something stringified'
>>> repr(something)
"Something repr'd"
>>>
```

Rather than having these methods return completely unrelated strings, the normal difference is that `repr` should (if possible) return a string that can be `eval`'d to create the equivalent object. This is why the quotes and backslashes turned up in the `repr` of the string. It's common for classes to have a `repr` of the form `ClassName(arguments)` and a different `str` form.

In this interactive session, we've been making the difference between `str` and `repr` explicit. This difference also crops up implicitly—which, incidentally, is why it's nicer to have these methods as protocol methods rather than forcing you to call them explicitly.

When you print an object, Python will use `__str__` to get a string representation. When you display an object at the interactive interpreter, `__repr__` is used.

```
>>> print something
Something stringified
>>> something
"Something repr'd"
```

When you use string interpolation, you can choose which method you want. %s is the formatting character for stringification, and %r will use `repr`.

```
>>> '%s and %r' % (something, something)
"Something stringified and Something repr'd"
```

If `__str__` is unavailable but `__repr__` is available, then `__repr__` will be used in its place. If you want both stringification and `repr`'ing to be the same, and want to implement only one of these methods, then you should choose `__repr__`.

Just for completeness, we should mention `__unicode__`. This protocol method is called when you ask for a Unicode representation of an object by calling `unicode` on it. Because strings are already Unicode in IronPython, it's not likely that you'll need it, but you may encounter it in libraries and existing Python code.

NOTE In fact, in IronPython 2, even if you explicitly call `unicode(something)`, the `__unicode__` method won't be called. This is really a bug, but arises because the `unicode` and `str` types are *both aliases to* `System.String`.

A common reason for providing these methods is for debugging; being able to get useful information about your objects in tracebacks can be invaluable. A method you're likely to implement to influence the way your objects behave in code is the `__nonzero__` method.

TRUTH TESTING OF OBJECTS

If you recall from the tutorial, `None`, empty strings, zero, and empty containers evaluate to `False` in logical expressions (and when used in `if` and `while` statements). Arbitrary objects evaluate to `True`. If you're implementing custom containers, then for consistency they should evaluate to `False` when they have no members; but, by default, they will *always* evaluate as `True`.

You can control whether an object evaluates to `True` or `False` by implementing the `__nonzero__` method. If this returns `True`, then the object will evaluate to `True`.

NOTE The full semantics of truth value testing are slightly more complicated than we've described so far. If an object doesn't implement `__nonzero__`, then Python will also check for `__len__`. If `__len__` is available and returns 0, then the object will evaluate to `False`. The `__nonzero__` method is useful for directly providing truth value testing without being dependent on supporting length.

```
>>> class Something(object):
...     def __nonzero__(self):
...         return False
...
>>> something = Something()
>>> bool(something)
False
```

These examples of working with magic methods have been nice and straightforward, almost trivially easy. This is good because even the more complex ones work on the same principles; you just need to know the rules (or know where to look them up). We now look at some more advanced protocols in the guise of operator overloading.

8.1.2 Operator overloading

Operator overloading allows you to control how objects take part in operations such as addition, subtraction, multiplication, and so on. You can create custom datatypes that represent values and can be used in calculations. You might want to define types that represent different currencies, and use a converter when different currencies are added together.

Each of the built-in operators (+, -, /, *, and friends) has an equivalent magic method that you can implement to support that operation. The method for addition is `__add__`. This method takes one argument: the object being added to. Traditionally, this argument is called `other`. Here's a simple class that behaves like the number three in addition:

```
>>> class Three(object):
...     def __add__(self, other):
```

```
...         return other + 3
...
>>> t = Three()
>>> t + 2
5
```

That's nice and easy. What happens if you try a slightly different addition with this class?

```
>>> 2 + t
Traceback (most recent call last):
  File , line 0, in ##29
TypeError: unsupported operand type(s) for +: 'int' and 'Three'
```

Because the `Three` instance is on the right-hand side of the addition, the integer add method is called—and integers don't know how to add themselves to the custom class. You can solve this problem with the `__radd__` method, which is the right-handed companion of `__add__` and the method called when a custom type is on the right-hand side of an addition operation.

```
>>> class Three(object):
...     def __add__(self, other):
...         return other + 3
...     def __radd__(self, other):
...         return 3 + other
...
>>> t = Three()
>>> t + 2
5
>>> 2 + t
5
```

The good news is that fixing the problem is this simple; the bad news is that you need to implement two methods for each operation. The next listing shows a simple way around this two-methods problem.

In-place operators

In-place operations are a convenient way of updating variables. You can easily increment a loop variable like this:

```
i += 1
```

In-place operations also have a protocol method, `__iadd__` in the case of addition. Numbers are immutable, so the underlying value that `i` points to isn't changed; instead, the addition is done, and the reference is bound to the new value. This operation is effectively *identical* to the following:

```
i = i + 1
```

You don't need to implement the in-place protocol methods for your types to support these operations. Implementing them allows you to customize them if you want to. Creating an event hook that supports in-place addition and subtraction to add and remove event handlers is one place that these methods are useful.

Table 8.1 shows the common[3] operator magic methods and the operations they provide. All the methods listed in this table also have right-hand and in-place equivalents as well.

Table 8.1 The methods used to emulate numeric types

Method	Operator	Operation
`__add__`	+	Addition
`__sub__`	-	Subtraction
`__mul__`	*	Multiplication
`__floordiv`	//	Floor division, truncating the result to the next integer down
`__div__`	/	Division
`__truediv`	/	Used for division when true division is on
`__mod__`	%	Modulo
`__divmod__`		Supports the built-in function `divmod`
`__pow__`	**	Raises to the power of
`__lshift__`	<<	Shifts to the left
`__rshift__`	>>	Shifts to the right
`__and__`	&	Bitwise AND
`__xor__`	^	Bitwise exclusive OR
`__or__`	|	Bitwise OR

A practical application of operator overloading is creating custom datatypes. The next listing is a datatype that allows you to store values with a known amount of uncertainty. When you add two uncertain values, the uncertainty is added. When you multiply them, the uncertainty increases dramatically! Listing 8.3 shows the code for an `Uncertainty` class that supports addition and multiplication.

Listing 8.3 Specifying length of custom containers

```
class Uncertainty(object):
    def __init__(self, value, spread):
        self.value = value
        self.spread = spread

    def __add__(self, other):
        if isinstance(other, Uncertainty):
            value = self.value + other.value
```

Handle adding to another Uncertainty instance

[3] For a complete list of these methods, including some of the less common ones, see http://docs.python.org/ref/numeric-types.html.

```
            spread = self.spread + other.spread
            return Uncertainty(value, spread)
        return Uncertainty(self.value + other, self.spread)

    __radd__ = __add__

    def __mul__(self, other):
        if isinstance(other, Uncertainty):
            value = self.value * other.value
            spread = (
                self.spread * other.spread +
                self.value * other.spread +
                self.spread * other.value
            )
            return Uncertainty(value, spread)
        return Uncertainty(self.value * other, self.spread * other)

    def __rmul__(self, other):
        return self.__mul__(other)

    def __repr__(self):
        return 'Uncertainty(%s, %s)' % (self.value, self.spread)

    def __str__(self):
        return u"%s\u00b1%s" % (self.value, self.spread)
```

Returns new instance

__radd__ does same as __add__

There are a few things of note in listing 8.3. Like integers and floating point numbers, `Uncertainty` instances are immutable, so addition and multiplication return new instances rather than modifying themselves. It would be an odd side effect if taking part in an addition modified a number that you held a reference to elsewhere! Inside the numeric protocol methods, `Uncertainty` has to handle operations involving another `Uncertainty` differently. The first thing it does is a type check. Because `__radd__` needs to do exactly the same as `__add__`, you make them the same method with an assignment.

There are also different implementations for `__repr__` and `__str__`. The `repr` of an `Uncertainty` looks like `Uncertainty(6, 3)`; the `str` (which is more of a pretty print) looks like `6±3`. `__str__` returns a Unicode string because it uses a Unicode character. This would cause problems in CPython, unless your default encoding can handle this character, but is fine under IronPython because strings are always Unicode. You'll still need a terminal (or GUI control) capable of displaying Unicode characters to see the fancy string representation.

Operator overloading isn't limited strictly to emulating numeric types. You can also implement these methods for objects that don't represent values, using ordinary Python syntax to perform operations on them. For example, the popular `path` module by Jason Orendorff[4] overloads division so that you can concatenate paths to strings by using the division operator, which is the normal path separator on Unix systems.

```
newPath = path('some path') / 'subdirectory' / 'filename'
```

[4] See http://pypi.python.org/pypi/path.py.

This kind of overloading operators with new meanings is a dubious practice in our opinion, but some programmers are fond of this kind of trick. Something that isn't dubious, and in fact is central to Python, is working with iterators—the subject of the next section.

8.1.3 Iteration

Iteration means repeatedly performing an action on members of an object. It's one of the fundamental concepts of programming, so being able to make your own objects iterable (or enumerable) is important.

You saw in the `Document` class for MultiDoc that providing a sequence-like API (a zero-indexed `__getitem__` method) gets you iteration for free. Life is rarely that simple, so you need to know how to support the iteration protocol.

When Python encounters iteration over an object—as a result of a `for` loop, a list comprehension, or an explicit call to the `iter` function—it will attempt to call `__iter__` on the object. If the object supports iteration, then this method should return an iterator. An iterator is an object that obeys the following three rules:

- It has a `__iter__` method that returns itself.[5]
- It has a `next` method that returns the next value from the iterator.
- When the iterator is consumed (all the values have been returned), calling `next` should raise the `StopIteration` exception.

That's all nice and easy, but you probably immediately noticed that, unlike the other protocol methods that we've used, `next` isn't held in the double embrace of underscores. Most magic methods are rarely called directly. The double underscores are a warning sign that, if you're calling them directly, then you should be sure you know what you're doing. Calling `next` on an iterator is perfectly normal. One place where we use this is for incrementing a counter, often on an iterator, like the following, returned by the built-in function `xrange`:

```
>>> counter = xrange(100)
>>> counter.next()
0
```

So without further ado, listing 8.4 demonstrates iteration with a simple class that returns every number between a start and a stop value.

Listing 8.4 An example `Iterator` class

```
class Iterator(object):
    def __init__(self, start, stop):
        self.stop = stop
        self.count = start

    def __iter__(self):      <-- Needed for iterator
        return self
```

[5] This allows an object to be its own iterator.

```
    def next(self):                     ◁— Returns next in sequence
        if self.count > self.stop:
            raise StopIteration
        count = self.count
        self.count += 1
        return count
>>> for entry in Iterator(17, 20):
...     print entry,                    ◁— Prints on one line
...
17 18 19 20
```

This example is a simple one, but your classes can support iteration by returning an object like this from *their* `__iter__` methods. Your iterator is free to do as much dynamic magic as it wants in `next`, such as reading from a file or a socket. A simpler, and more powerful, way of implementing an iterator is through generators.

8.1.4 Generators

In the iterator created in listing 8.4, you needed to store state, in the form of the `count` instance variable, so that successive calls to next could calculate the next return value. A simpler pattern is available using the Python `yield` keyword, which is similar to the C# `Yield Return` statement.

When a function (or method) uses `yield`, it becomes a generator. Generators are a lightweight form of coroutines.

Coroutines and generators

Coroutines are subroutines that have multiple entry points and allow you to suspend and resume execution at certain points.

Python generators are iterators that suspend execution (and return a value) at a `yield` statement. On the next iteration, execution resumes at the point it was suspended, making them a lightweight form of coroutines.

Features added in Python 2.5 bring generators closer to full coroutines. See PEP 342 for the details.[6]

Generators are a simple, but powerful, language feature. The BBC Research & Development department has used them to implement a new form of concurrency powerful enough to stream video and audio: the Kamaelia project.[7]

Where a function uses `yield`, the value is returned and execution of the generator is suspended. On the next iteration, execution continues at the point it was suspended. Complex iteration can be implemented with generators, without having to explicitly store state.

---

[6] A PEP is a Python Enhancement Proposal, and they're the way new features are proposed for Python. PEP 342 (now part of Python 2.5) lives at http://www.python.org/dev/peps/pep-0342/.

[7] See http://www.kamaelia.org/Home.

Listing 8.5 is an example with a `Directory` object that stores all the files in a specified path from the filesystem. It recurses into subdirectories, storing them as `Directory` objects. The `__iter__` method is implemented as a generator and also recurses through subdirectories.

Listing 8.5 `Directory` class that supports iteration with a generator

```
import os

class Directory(object):
    def __init__(self, directory):
        self.directory = directory
        self.files = []
        self.dirs = []

        for entry in os.listdir(directory):        <-- Returns all paths in directory
            full_path = os.path.join(directory, entry)
            if os.path.isfile(full_path):
                self.files.append(entry)
            elif os.path.isdir(full_path):
                self.dirs.append(Directory(full_path))

    def __iter__(self):
        for entry in self.files:
            yield os.path.join(self.directory, entry)     <-- Yields files first

        for directory in self.dirs:
            for entry in directory:
                yield entry        <-- Recursively iterates over subdirectories

>>> for entry in Directory(some_path):
...     print entry
...
C:\\some_path\\file1.txt
C:\\some_path\\subdirectory\\another_file.txt
(and so on...)
```

Because the `__iter__` method yields, it returns a generator, which is a specific kind of iterator. As an iterator, it has the `next` method available.

```
>>> generator = iter(Directory(some_path))
>>> generator.next()
'C:\\some_path\\some_file.txt'
```

You can easily extract all members from objects that support iteration, by calling `list` (or `tuple`) on them.

```
>>> paths = list(Directory(some_path))
```

We can't leave Python protocols without looking at one of the most common programming concepts implemented with magic methods: equality and inequality.

8.1.5 Equality and inequality

Comparing objects is a basic part of programming and one of the first things a new programmer will learn; you need to know how to support equality and inequality operations on your own classes. You do this with the `__eq__` (equals) and `__ne__` (not

equals) protocol methods. For equality to work correctly, you need to implement *both* these methods. If you don't provide these methods, then Python will use object identity as the test for equality. An object will only compare equal to itself and not equal to everything else.

Listing 8.6 is a class with custom equality and inequality methods defined.

Listing 8.6 Object equality and inequality methods

```
class Value(object):
    def __init__(self, value):
        self.value = value

    def __eq__(self, other):                    # Invoked for equality
        if isinstance(other, Value):            # When comparing against Value objects
            return self.value == other.value
        return False

    def __ne__(self, other):                    # Invoked for inequality
        return not self.__eq__(other)

if __name__ == '__main__':
    v1 = Value(6)
    v2 = Value(3)
    v3 = Value(6)
    assert v1 == v3
    assert v1 != v2
 assert v1 != 6
```

The `__ne__` method returns the `not` of the equality method. This is the simplest possible implementation, and it would be nice if Python did this for you! The `__eq__` method always returns `False` unless it's compared against another `Value` instance. If it's compared to a `Value` object, then it compares `value` attributes. Without the `isinstance` check, equality testing would fail with an attribute error for objects that don't have a `value` attribute.

Equality and hashing

Hashing is the way that objects are stored in dictionaries. If a class defines equality (or comparison) methods, then it *should* define a `__hash__` method so that instances can be hashed correctly. The rule is that objects that compare equal should hash equally. The exception to this rule is that mutable objects should be unhashable and raise a `TypeError`. Custom `__hash__` methods typically use the built-in `hash` function.

The `__hash__` method for our `Value` class would be

```
def __hash__(self):
    return hash(self.value)
```

The `__hash__` method for an unhashable object would be as follows:

```
def __hash__(self):
    raise TypeError('Value objects are unhashable')
```

As well as protocols to control equality and inequality comparisons, there are protocol methods for the other comparison operators.[8] Table 8.2 shows the rich comparison operators and corresponding protocol methods.

Method	Operator	Operation
`__eq__`	==	Equality
`__ne__`	!=	Inequality
`__lt__`	<	Less than
`__le__`	<=	Less than or equal to
`__gt__`	>	Greater than
`__ge__`	>=	Greater than or equal to

Table 8.2 The methods used for rich comparison

To support the full range of rich comparison operators, you need to implement all these methods.[9]

TIP Another, easier way of supporting all comparison operations on an object is the `__cmp__` method, which is used to overload the `cmp` built-in function. This method should return a negative integer if the object is less than the one it's being compared with, zero if it's equal to the other object, and a positive integer if it's greater than the other object. With `__cmp__` you don't know which comparison operation is being performed, so it's less flexible than overloading individual rich comparison operators.

We've now looked at quite a range of Python's protocol methods. We haven't used them all (by any stretch of the imagination), but we've covered all the most common ones. The important thing is to know the ones that you'll use most often and where to look up the rest!

The next section will look at two protocol methods that are a little different. Instead of enabling a specific operation, `__getattr__` and friends allow you to customize attribute access.

8.2 Dynamic attribute access

Python attribute access uses straightforward syntax, shared with other imperative languages like Java and C#. Through properties, you can control what happens when individual attributes are fetched or set; but, with Python, you can provide access to arbitrary attributes through the `__getattr__` method.

---

8 This page in the Python documentation lists (among other things) the protocol methods for rich comparison: http://docs.python.org/ref/customization.html.

9 See the following article on rich comparison in CPython and IronPython, which also shows a simple `Mixin` class to easily provide rich comparison just by implementing `__eq__` and `__lt__`: http://www.voidspace.org.uk/python/articles/comparison.shtml.

The flip side of this is being able to dynamically access attributes when you have their names stored as strings. Python supports this through a set of built-in functions.

8.2.1 Attribute access with built-in functions

The following are four built-in functions available for working with attributes:

- *`hasattr(object, name)`*—Checks whether an object has a specified attribute, returning `True` or `False`.
- *`getattr(object, name)`*—Fetches a named attribute from an object. If the attribute doesn't exist, then an `AttributeError` will be raised. `getattr` also takes an optional third argument. If this is supplied, it's returned as a default value when the attribute is missing (instead of raising an exception).
- *`setattr(object, name, value)`*—Sets the named attribute on an object with the specified value.
- *`delattr(object, name)`*—Deletes the named attribute from an object. If the attribute doesn't exist, then an `AttributeError` will be raised.

NOTE The attribute-access functions invoke the whole Python attribute lookup machinery. Underneath is a dictionary per object[10] that stores attributes. This dictionary is available as `__dict__`. The predominance of dictionaries in the implementation of Python demonstrates how important the concept of namespaces is to the language. A dictionary is a namespace, mapping names to objects. Classes, instances, and modules are all examples of namespaces and are all implemented using dictionaries (which are exposed through `__dict__`).

So what are these functions useful for?

`hasattr` in particular is useful as a duck typing mechanism. If you want to support a particular interface, one possible pattern is *it's better to ask forgiveness than ask permission.*

```
try:
    instance.someMethod()
except AttributeError:
    # different kind of object
```

Sometimes you *want* to ask permission, though—perhaps particular objects need different treatment. You could do type checking with `isinstance`, but this defeats duck typing and requires users of your code to use specific types instead of passing in objects with a compatible API. Instead, you can use `hasattr`, as follows:

```
if hasattr(instance, 'someMethod'):
    instance.someMethod()
    # and so on
```

`getattr`, `setattr`, and `delattr` are particularly useful when you have a list of attributes as strings (potentially as the result of a call to `dir`) and need to loop over the list

[10] For creating massive numbers of objects, you can avoid the overhead of a dictionary per object, by using a memory optimisation mechanism called `__slots__` for storing named members only.

performing operations. In a brief while, we'll use these functions in a proxy class that illustrates the attribute-access protocol methods. Before we can do that, you need to learn about the attribute-access protocol methods themselves.

8.2.2 Attribute access through magic methods

As with built-in functions like `len` and `str`, the attribute-access functions have corresponding protocol methods that you can implement. With these functions, the relationship between the protocol method and the corresponding function is a little more complex.

The attribute-access functions invoke the full attribute lookup mechanism for Python objects. Included in this mechanism are three protocol methods that you can implement to govern how *some* attributes are fetched, set, and deleted.

The three protocol methods (`hasattr` has no direct equivalent as a protocol method) are as follows:

- *`__getattr__(self, name)`*—Provides the named attribute. This method is called *only* for attributes that aren't found by the normal lookup machinery.
- *`__setattr__(self, name, value)`*—Sets the named attribute to the specified value. This method is invoked for all instance attribute setting.
- *`__delattr__(self, name)`*—Deletes the named attribute. This method is invoked for all instance attribute deletion.

As you can see, these methods are asymmetrical. `__setattr__` and `__delattr__` are invoked for all set and delete operations on an instance, whereas `__getattr__` is only invoked if the attribute doesn't exist. According to the Python documentation,[11] this is for efficiency and to allow `__setattr__` to access instance variables.

At Resolver Systems, we've created a spreadsheet application with IronPython.[12] This is a spreadsheet for creating complex models or business applications using a spreadsheet interface. IronPython objects that represent the spreadsheet are exposed to the user, and can be manipulated from user code sections. One of our core classes is the `Worksheet`, representing different sheets in the spreadsheet. The user accesses values in a worksheet by indexing it with the column and the row number. Most spreadsheet users are more used to referring to locations by the A1 style name, so we use `__getattr__` to provide a convenient way of accessing values using a syntax like this: `worksheet.A1 = 3`. We implemented this with code that looks like the following:

```
def __getattr__(self, name):
    location = CoordinatesFromCellName(name)
    if location is None:
        raise AttributeError(name)
    col, row = location
    return self[col, row]
```

[11] See http://docs.python.org/ref/attribute-access.html.
[12] See http://www.resolversystems.com or http://www.resolverhacks.net.

We also need to implemented `__setattr__` to allow users to set values in the worksheet. `__setattr__` is called for *all* attribute-setting operations, so we needed to delegate to `object.__setattr__` for all attributes *except* cells.

```
def __setattr__(self, name, value):
    location = CoordinatesFromCellName(name)
    if location is not None:
        col, row = location
        self[col, row] = value
    else:
        object.__setattr__(self, name, value)
```

You can put these methods to work in a proxy class that proxies attribute access from one object to another.

8.2.3 Proxying attribute access

You've seen how Python has no true concept of private or protected members. If you come to Python from a language like C#, you'll probably be surprised at how infrequently you need these features. You can achieve the same effect as private members through a factory function and a proxy class like the one in listing 8.7.

Listing 8.7 Attribute protection with a factory function and proxy class

```
def GetProxy(thing, readlist=None,
             writelist=None, dellist=None):

    class Proxy(object):
        def __getattr__(self, name):
            if readlist and name in readlist:       <- Allows access only to names in readlist
                return getattr(thing, name)
            else:
                raise AttributeError(name)          <- Raises AttributeError for disallowed names

        def __setattr__(self, name, value):
            if writelist and name in writelist:
                setattr(thing, name, value)
            else:
                raise AttributeError(name)

        def __delattr__(self, name):
            if dellist and name in dellist:
                delattr(thing, name)
            else:
                raise AttributeError(name)

    return Proxy()
```

You pass in the object you want proxied as the `thing` argument to `GetProxy`. You also pass in three lists of attribute names (all optional). These are attributes that you do want to allow access to—for read access, write access, and delete access.

When you call `GetProxy`, it returns an instance of the `Proxy` class. This instance has access to `thing`, and the attribute lists, through the closure; but, from the instance, there's no way to get back to the original object. Accessing attributes on the `Proxy`

instance triggers `__getattr__` (or `__setattr__` or `__delattr__`). If the attribute name is in the corresponding attribute list, then the access is allowed. If the attribute isn't in the list, then the access is disallowed, and an `AttributeError` is raised.

This pattern is useful for more than protecting attribute access; it's a general example of the delegation pattern,[13] but it does have a couple of restrictions. Although it works fine for fetching normal methods, magic methods are looked up on the class rather than the instance. Indexing the object, or any other operation that uses a protocol method, will look for the method on the `Proxy` class instead of the original instance. Two possible approaches to solving this are as follows:

- You can provide the magic methods you want on the `Proxy` class and proxy those as well.
- Because magic methods are looked up on the class, you can provide a metaclass that has `__getattr__` and friends implemented and does the proxying.

Another restriction is that, although attribute access is proxied, the proxy instance has a different type to the object it's proxying and so can't necessarily be used in all the same circumstances as the original.

Using the `__getattr__` and `__setattr__` protocol methods to customize attribute access isn't something you do every day; but, in the right places, they can be used to create elegant and intuitive APIs.

Something that takes us even deeper into the dynamic aspects of Python is metaprogramming with Python metaclasses. Understanding metaclasses will further deepen your understanding of Python and open up some fun possibilities.

8.3 Metaprogramming

Metaprogramming is the programming language or runtime assisting with the writing or modification of your program. The classic example is runtime code generation. In Python and IronPython, this is supported through the `exec` statement and built-in compile/eval functions. Python source code is text, so generating code using string manipulation and then executing it is relatively easy.

But, code generation has to be relatively deterministic (your code that generates the strings is going to be following a set of rules that you determine), meaning that it's usually easier to provide objects rather than go through the intermediate step of generating and executing code. An exception is when you generate code from user input, perhaps by translating a domain-specific language into Python. This is the approach taken by the Resolver One spreadsheet, which translates a Python-like formula language into Python expressions.

Python has further support for metaprogramming through something called metaclasses. These allow you to customize class creation—that is, modify classes or perform actions at the point at which they're defined. Metaclasses have a reputation for being deep black magic.

---

[13] See http://en.wikipedia.org/wiki/Delegation_pattern.

> *Metaclasses are deeper magic than 99% of users should ever worry about. If you wonder whether you need them, you don't.*
>
> —Python Guru Tim Peters

They're seen as magic because they can modify code away from the point at which it appears in your source. In fact, the basic principles are simple to grasp, and no good book on Python would be complete without an introduction to them.

8.3.1 Introduction to metaclasses

In Python, everything is an object. Functions and classes are all first-class objects that can be created at runtime and passed around your code. Every object has a type. For most objects, their type is their class, so what's the type of a class? The answer is that classes are instances of their metaclass.

Just as objects are created by calling their class with the correct parameters, classes are created by calling their metaclass with certain parameters. The default metaclass (the type of classes) is `type`. This leads to the following wonderful expression:

```
type(type) is type
```

`type` is itself a class, so *its* type is `type`![14]

What does this have to do with metaprogramming? Python is an interpreted language—classes are created at runtime; and, by providing a custom metaclass, you can control what happens when a class is created, including modifying the class.

As usual, the easiest way to explain this is to show it in action. Listing 8.8 shows the simplest possible metaclass.

Listing 8.8 The simplest example of a metaclass

```
class PointlessMetaclass(type):                  <— Metaclasses inherit from type
    def __new__(meta, name, bases, classDict):
        return type.__new__(meta, name, bases, classDict)

class SomeClass(object):
    __metaclass__ = PointlessMetaclass          <— Set the metaclass on the class
```

This metaclass doesn't *do* anything, but illustrates the basics of the metaclass. You set the metaclass on a class by assigning the `__metaclass__` attribute inside the class definition. Subclasses automatically inherit the metaclass of their superclass.[15] When the class is created, the metaclass is called with a set of arguments. These are the class name, a tuple of base classes, and a dictionary of all the attributes (including methods) defined in the class. To customize class creation with a metaclass, you need to inherit from `type` and override the `__new__` method.

You can experiment here by defining methods and attributes on `SomeClass`, and putting print statements in the body of `PointlessMetaclass`. You'll see that methods appear as functions in the `classDict`, keyed by their name. Inside the metaclass, you

---

[14] This isn't true in IronPython 1—which is a bug that has been fixed in IronPython 2.

[15] Meaning that you can't have incompatible metaclasses where you have multiple inheritance.

The __new__ constructor

So far, we've talked about `__init__` as the constructor method of objects. `__init__` receives `self` as the first argument, the freshly created instance, and initializes it.

The instance that `__init__` receives is created by `__new__`, so it's technically more correct to call `__new__` the constructor and `__init__` an initializer. `__new__` receives the class as the first argument, plus any additional arguments used in the construction. (It receives the same arguments as `__init__`.)

`__new__` is needed for creating immutable values; setting the value in `__init__` would mean that values could be modified simply by calling `__init__` on an instance again. You can subclass the built-in types in Python; but, to customize immutable values like `str` and `int`, you need to override `__new__` rather than `__init__`.

can modify this dictionary (plus the `name` and the `bases` tuple if you want) to change how the class is created. But what can you use metaclasses for?

8.3.2 Uses of metaclasses

Despite their reputation for being deep magic, sometimes only a metaclass can achieve something that's difficult or impossible to do via other means. They're invoked at class-creation time, so they can be used to perform operations that would otherwise require a manual step. These operations include[16] the following:

- *Registering classes as they're created*—Often done by database Object-Relational Mapping (ORM) because the classes you create relate to the shape of the database table you'll interact with. Registering classes as they're created can also be useful for autoregistering plugin classes.
- *Enabling new coding techniques*—Such as enabling a declarative way of declaring database schema, as is the case for the Elixir[17] ORM framework.
- *Providing interface registration*—Includes autodiscovery of features and adaptation.
- *Class verification*—Prevents subclassing or verifies code quality, such as checking that all methods have docstrings or that classes meet a particular standard.
- *Decorating all the methods in a class*—Can be useful for logging, tracing, and profiling purposes.
- *Mixing in appropriate methods without having to use inheritance*—Can be one way of avoiding multiple inheritance. You can also load in methods from non-code definitions—for example, by loading XML to create classes.

[16] This borrows some of the use cases for metaclasses from an excellent presentation by Mike C. Fletcher. See http://www.vrplumber.com/programming/metaclasses-pycon.pdf.

[17] Elixir is a declarative layer built on top of the popular Python ORM SQLAlchemy. See http://elixir.ematia.de/trac/wiki.

We can show a practical use of metaclasses with a profiling metaclass. This is an example of the fifth use of metaclasses listed—wrapping every method in a class with a decorator function.

8.3.3 A profiling metaclass

One of the use cases mentioned in the bulleted list is wrapping all methods with a decorator. You can use this for profiling by recording method calls and how long they take. This approach is useful if you're looking to optimize the performance of your application. With IronPython you always have the option of moving parts of your code into C# to improve performance; but, before you consider this, it's important to profile so that you know exactly which parts of your code are the bottlenecks—the results are often not what you'd expect. At Resolver, we've been through this process many times, and have always managed to improve performance by optimizing our Python code; we haven't had to drop down into C# to improve speed so far.

For profiling IronPython code you can use the .NET 'DateTime'[18] class.

```
from System import DateTime
start = DateTime.Now

someFunction()

timeTaken = (DateTime.Now - start).TotalMilliseconds
```

There's a drawback to this code. `DateTime` has a granularity of about 15 milliseconds. For timing individual calls to fast-running code, this can be way too coarse. An alternative is to use a high-performance timer class from `System.Diagnostics`: the `Stopwatch`[19] class. Listing 8.9 is the code to time a function call using a `Stopwatch`.[20]

Listing 8.9 Timing a function call with the `Stopwatch` class

```
from System.Diagnostics import Stopwatch
s = Stopwatch()
s.Start()

someFunction()

s.Stop()
timeTaken = s.ElapsedMilliseconds
```

You can wrap this code in a decorator that tracks the number of calls to functions, and how long each call takes (listing 8.10).

Listing 8.10 A function decorator that times calls

```
from System.Diagnostics import Stopwatch

timer = Stopwatch()
times = {}          <-- Cache to store times
```

[18] See http://msdn2.microsoft.com/en-us/library/system.datetime.aspx.

[19] See http://msdn2.microsoft.com/en-us/library/system.diagnostics.stopwatch(vs.80).aspx. Note that, on multicore AMD processors, the `StopWatch` can sometimes return negative numbers (appearing to travel backwards in time!). The solution is to apply this fix: http://support.microsoft.com/?id=896256.

[20] The Python standard library `time.clock()` is implemented using `StopWatch` under the hood.

```
def profiler(function):                          Decorator which
    def wrapped(*args, **keywargs):              wraps functions
        if not timer.IsRunning:
            timer.Start()

        start = timer.ElapsedMilliseconds
        retVal = function(*args, **keywargs)
        timeTaken = timer.ElapsedMilliseconds - start

        name = function.__name__
        function_times = times.setdefault(name, [])
        function_times.append(timeTaken)
        return retVal                                    Adds current duration
    return wrapped
```

The profiler decorator takes a function and returns a wrapped function that times how long the call takes. The times are stored in a cache (the `times` dictionary), keyed by the function name.

> **Optimizing IronPython code**
>
> It's *always* important to profile code when optimizing. The bottlenecks are rarely quite where you expect them, and you can find effective ways of speeding up your code only if you know exactly which bits are slow.
>
> For experienced Python programmers, optimizing IronPython code is *especially* important because the performance of IronPython is so different from CPython. For example, function calls are much less expensive, as are arithmetic and operations involving the primitive types. On the other hand, operations with some of the Python types like tuples and dictionaries can be more expensive.[21]

Now you need a metaclass that can apply this decorator to all the methods in a class. You'll be able to recognize methods by using `FunctionType` from the Python standard library `types` module. Listing 8.11 shows a metaclass that does this.

TIP Don't forget that, to import from the Python standard library, it needs to be on your path. This happens automatically if you installed IronPython from the msi installer. Otherwise, you need the Python standard library (the easiest way to obtain it is to install the appropriate version of Python) and to set the path to the library in the `IRONPYTHONPATH` environment variable.

Listing 8.11 A profiling metaclass that wraps methods with `profiler`

```
from types import FunctionType                   Or FunctionType =
                                                 type(some_function)
class ProfilingMetaclass(type):

    def __new__(meta, classname, bases, classDict):
```

[21] Which is hardly surprising—the Python built-in types are written in C and have been fine-tuned over the last fifteen years or more. Their implementation in IronPython is barely a handful of years old.

```
        for name, item in classDict.items():          Checks for methods
            if isinstance(item, FunctionType):    <-
                classDict[name] = profiler(item)    <- Wraps methods with profiler
        return type.__new__(meta, classname, bases, classDict)
```

Of course, having created a profiling metaclass, we need to use it. Listing 8.12 shows how to use the `ProfilingMetaclass`.

Listing 8.12 Timing method calls on objects with `ProfilingMetaclass`

```
from System.Threading import Thread

class Test(object):

    __metaclass__ = ProfilingMetaclass

    def __init__(self):
        counter = 0
        while counter < 100:
            counter += 1
            self.method()

    def method(self):
        Thread.CurrentThread.Join(20)

t = Test()

for name, calls in times.items():
    print 'Function: %s' % name
    print 'Called: %s times' % len(calls)
    print ('Total time taken: %s seconds' %
               (sum(calls) / 1000.0))
    avg = (sum(calls) / float(len(calls)))
    print 'Max: %sms, Min: %sms, Avg: %sms' % (max(calls), min(calls), avg)
```

When the `Test` class is created, all the methods are wrapped with the profiler. When it's constructed, it calls `method` 100 hundred times. When the code has run, you print out the entries in the `times` cache to analyze the results. It should print something like this:

```
Function: method
Called: 100 times
Total time taken: 2.07 seconds
Max: 39ms, Min: 16ms, Avg: 20.7ms

Function: __init__
Called: 1 times
Total time taken: 2.093 seconds
Max: 2093ms, Min: 2093ms, Avg: 2093.0ms
```

If this were real code, you'd know how many calls were being made to each method and how long they were taking. With this simple technique, the `times` cache includes calls to other methods, as well as overhead for the profiling code itself. It's possible to write more sophisticated profiling code that tracks trees of calls so that you can see how long different branches of your code take.

Metaclasses are magic, but perhaps not as black as they're painted. This is true of all the Python magic that you've been learning about. Python's magic methods are magic because they're invoked by the interpreter on your behalf, not because they're difficult to understand. Using these protocol methods is a normal part of programming in

Python. You should now be confident in navigating which methods correspond to language features and how to implement them.

We've plunged into the depths of Python, so now it's time to take a journey around the edges—in particular, the edges where IronPython meets the Common Language Runtime (CLR). Not all the key concepts in the CLR match directly to Python semantics, particularly because it's a dynamic language. To make full use of framework classes, you need to know a few details.

8.4 IronPython and the CLR

So far, we've focused on navigating the .NET framework and the Python language, emphasizing the skills you need to make effective use of IronPython. But not all .NET concepts map easily to Python concepts—certain corners of .NET can't be brought directly into Python. This section looks at some of these dusty corners and how they work in IronPython. Most of the things we explore are simple; but, even if they're easy, you need to know the trick in order to use them.

The first line of IronPython/CLR integration is the `clr` module; in several of the corners we look in, you'll find new `clr` module functionality. As well as covering some interesting topics, this section will be a useful reference as you encounter these situations in the wild.

> **Delegates and CallTarget0**
>
> One aspect of IronPython and .NET interoperation that you're likely to need to know about, but doesn't fit into any of the subsections here, is the `CallTarget` family of delegates. They're useful for explicitly creating a delegate when IronPython can't magically create one for you.[22] You used this in the last chapter when invoking a callable onto the GUI thread with Windows Forms.
>
> Several delegates correspond to the number of arguments your callable needs to take (`CallTarget0` for zero arguments, `CallTarget1` for a single argument, and so on). We find it simpler to use `CallTarget0` and wrap the callable with a lambda that calls it with any arguments.
>
> In IronPython 1, `CallTarget0` lives in the `IronPython.Runtime.Calls` namespace. In IronPython 2, it lives in `IronPython.Compiler`.

The first dusty corner that we peer into is .NET arrays.

8.4.1 .NET arrays

Arrays are one of the standard .NET container types, roughly equivalent to the Python tuple.[23] They're also statically typed, of course. When you create an array, you specify type and the size. The array can then be populated only with members of the specified type.

---

[22] If you're using .NET 3.0, you can use the `Action` and `Func` delegates instead.

[23] Although arrays allow you to change their contents (they're mutable). Like tuples, they have a fixed size.

The tuple is a built-in Python type; and, being dynamically typed, it's generally more convenient to use than arrays when you have a choice. When you're working with .NET APIs, they will sometimes *require* an array, and you need to know how to create them.

In C#, you can create an array of integers using the syntax in figure 8.1.

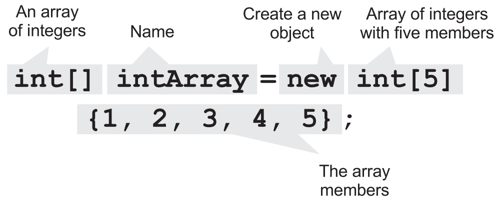


Figure 8.1 Creating an integer array in C#

In IronPython, you use the following syntax to achieve the same thing:

```
>>> from System import Array
>>> intArray = Array[int]((1, 2, 3, 4, 5))
```

This syntax looks like you're indexing the Array type, so it's valid Python syntax. This tells IronPython that you want an array of integers. It's populated from a tuple of the initial members.

> **Generics**
>
> Arrays are far from the only place where you need to specify a type when working with the .NET framework. The realm of generics is another of these places. Generics allow .NET types to work with any type. A good example is the `List` type from `System.Collections.Generic`, which is similar to the Python list.
>
> In C#, you'd use syntax like the following to create a `List` that holds strings:
>
> ```
> List<string> dinosaurs = new List<string>();
> ```
>
> IronPython reuses the indexing syntax for generics, so the IronPython equivalent would be the following:
>
> ```
> dinosaurs = List[str]()
> ```
>
> Unfortunately, in some cases (such as overloaded methods) IronPython can't tell whether you're specifying the generic option, or the typed option. In this case, IronPython will always pick the typed variant. The IronPython team is working to see if they can find some way of specifying the generic option without having to introduce new syntax.

This technique isn't the only way to create arrays. Another way is by using the `CreateInstance` class method, which can be used to create multidimensional arrays.

CREATEINSTANCE AND MULTIDIMENSIONAL ARRAYS

The `Array` type also has a `CreateInstance` class method for constructing arrays. Using this method doesn't require you to prepopulate the array; it will be initialized with a default value for every member. For integer arrays, this will be 0; for reference types, it

will be `null` (`None`). There are several different ways of calling `CreateInstance`.[24] The most common are the following:

```
>>> multiArray = Array.CreateInstance(str, 5)
```

This code creates an array of strings with five members. The next way creates a two-dimensional array from a type and two integers that specify the size of each dimension.

```
>>> multiArray = Array.CreateInstance(str, 5, 5)
```

This creates a two-dimensional array with 5 x 5 members. You can access individual members in this array using another extension to Python syntax.

```
>>> aMember = multiArray[2, 2]
```

In normal Python syntax, this would be indexing `multiArray` with the tuple `(2, 2)`. Instead, you're accessing the member at location 2, 2. (If you try indexing with a tuple, it will fail.)[25] Because this is valid Python syntax, no changes to the language were necessary to support indexing multidimensional arrays.

You can create arrays with any number of dimensions by passing a type and an integer array specifying the size of each dimension. The number of members of the array determines the number of dimensions. The following code creates a five-dimensional array of `objects`, with five members in each dimension:

```
>>> dimensions = Array[int]((5, 5, 5, 5, 5))
>>> multiArray = Array.CreateInstance(object, dimensions)
>>> multiArray[0, 0, 0, 0, 0] = SomeObject()
>>> print multiArray[0, 0, 0, 0, 0]
<SomeObject object at 0x....>
```

Another way of creating multidimensional arrays is by creating an array of arrays—the so-called *jagged array*. The following code creates an array with five members; each member is an array of five members:

```
>>> arrays = tuple(Array[int]((0, 0, 0, 0, 0)) for i in range(5))
>>> multiArray = Array[Array[int]](arrays)
>>> multiArray[0][0]
0
```

As you can see, this uses slightly different indexing syntax. Indexing the array returns the member at that index, which itself is an array. You then index that to retrieve a specific member.

Arrays have useful methods and properties such as `Copy`, to copy an array, or `Rank`, which tells you how many dimensions an array has. If you're going to be using arrays a lot, it's worthwhile to familiarize yourself with the array API. The next corner of .NET we shine a light on is overloaded methods.

---

[24] For a list of the different ways to call `CreateInstance`, see http://msdn2.microsoft.com/en-us/library/system.array.createinstance.aspx.

[25] In his technical review, Dino noted: "I think this is an IronPython bug—or at least a small deficiency in the way we map array to Python."

8.4.2 Overloaded methods

C# allows overloaded methods—the same method being defined several times, but with different types in the parameter list. The compiler can tell at *compile time* which of the overloads you're calling by the types of the arguments.

This largely works seamlessly in IronPython. The IronPython runtime uses reflection (the .NET equivalent of introspection using the `System.Reflection` namespace) to deduce which overload you're calling. If IronPython gets it wrong, it's a bug and you should report it on the IronPython mailing list or the issue tracker on the IronPython CodePlex site!

On occasion, you may want direct access to a specific overload. IronPython provides an `Overloads` collection that you can index by type (or a tuple of types). The IronPython documentation provides the following illustration, choosing the `WriteLine` method that takes an object and then passing `None` to it:

```
from System import Console
Console.WriteLine.Overloads[object](None)
```

Three more .NET concepts that don't map easily to Python are the `out`, `ref`, and pointer parameters.

8.4.3 out, ref, params, and pointer parameters

Occasionally you'll find a .NET method that requires an `out` or `ref` type parameter, an argument that takes a pointer, or one that uses the `params` keywords. These concepts don't exist directly in IronPython, but you can handle them all straightforwardly.

In C#, the body of the method modifies `out` (output) parameters. In Python, if you passed in an immutable value for this parameter, such as a string or an integer, then in Python the original reference wouldn't be modified—which would kind of defeat the purpose of having the parameter. This difficulty is easily solved in IronPython. Because the original value of an `out` parameter isn't used, the IronPython runtime does some magic, and the updated value becomes an extra return value. Let's observe this in action with the `Dictionary.TryGetValue` method.[26] This method takes two parameters: the key you're searching for and an `out` parameter modified to contain the corresponding value if it's found. Normally, it returns a Boolean that indicates whether or not the key was found. The `out` parameter is modified to contain the value. In IronPython, you get two return values.

```
>>> from System.Collections.Generic import Dictionary
>>> d = Dictionary[str, str]()
>>> d['key1'] = 'Value 1'
>>> d['key2'] = 'Value 2'
>>> d.TryGetValue('key1')
(True, 'Value 1')
>>> d.TryGetValue('key3')
(False, None)
```

[26] This is the example used by Haibo Luo in a blog entry. See http://blogs.msdn.com/haibo_luo/archive/2007/10/04/5284947.aspx.

Passing by reference is similar; in fact, you can often pass by reference instead of using an `out` parameter. To pass by reference, you need to create a reference using a type provided by the `clr` module. One example is the class method `Array.Resize` for resizing arrays. It's a generic method, so you need to provide type information—making it a great example of how dynamic typing makes life easier! This method takes a reference to an array and the new size as an integer.

NOTE `clr.Reference` isn't to be confused with `clr.References`. `clr.References` is a list of all the assemblies (actual assembly objects) that you've added a reference to.

First you construct an array of integers to resize, and then create a reference to that array. `clr.Reference` is a generic, so you need to tell it the type of the reference you require.

```
>>> from System import Array
>>> import clr
>>> array = Array[int]((0, 1, 2, 3, 4))
>>> array
Array[int](0, 1, 2, 3, 4)
>>> ref = clr.Reference[Array[int]](array)
```

Next you call `Array.Resize`. You need to specify the type of array (again); and, because `Resize` is a generic method, you need to specify the type of array here as well (otherwise the IronPython runtime won't be able to find the right target and will report `TypeError: no callable targets`). The resized array is stored as the `Value` attribute of the reference.

```
>>> Array[int].Resize[int](ref, 3)
>>> ref.Value
Array[int]((0, 1, 2))
```

A pointer is a good old-fashioned kind of reference. You usually come across them when dealing with unmanaged resources, and often they'll be pointers to integers. Integer pointers are represented with the `System.IntPtr` structure.

```
>>> from System import IntPtr
>>> ptr = IntPtr(3478)
>>> ptr.ToInt32()
3478
>>> IntPtr.Zero
<System.IntPtr object at 0x… [0]>
```

We've not encountered this in the wild; but, for completeness, we need to mention another way of passing arguments: the C# `params` keyword. This is used to pass a variable number of arguments (like the Python `*args` syntax), and the method receives a collection of the arguments you pass. In IronPython, .NET APIs that use the `params` keyword are handled seamlessly—you pass arguments as normal.

```
>>> something.Method(1, 2, 3, 4)
```

Another area of .NET that can potentially cause confusion is the peculiarity of *value types*.

8.4.4 Value types

The .NET framework makes a distinction between value types and reference types. The technical distinction is that value types are allocated on the stack or inline within an object, whereas reference types are allocated on the heap. Value types are passed by value (rather than by reference—hence, the names) making them efficient to use.[27] Table 8.3 lists the value and reference types.

Table 8.3 .NET value and reference types

Value types	Reference types
All numeric datatypes (including `Byte`)	Strings
`Boolean`, `Char`, and `Date`	Arrays (even arrays of value types)
All structures, even if members are reference types	Class types (like `Form` and `Object`)
Enumerations (because the underlying type is numeric)	Delegates

In .NET land, you can pass a value type to methods expecting a reference type. Take, for example, this overload of `Console.WriteLine`:

```
Console.WriteLine(String, Object)
```

This takes a format string and an object, and it writes the string representation of the object to the standard output stream. The declaration of `Object` as the second argument means that this method can take any arbitrary object, but also means that it's expecting a reference type (because `Object` is a reference type). If you call this method with a value type, like an integer, then the CLR will automatically box and unbox the value so that it can be used in place of a reference type.

Enumeration values and names

Enumerations (enums) are implemented in .NET with underlying numeric values (usually integers) representing the different enumeration flags. In C#, you can cast enumeration members to their underlying value. In IronPython, you can do the equivalent.

```
>>> from System.Net.Sockets import SocketOptionName
>>> int(SocketOptionName.UnblockSource)
18
```

If you want programmatic access to the name, then you can call `str` on the enumeration member.

```
>>> str(SocketOptionName.UnblockSource)
'UnblockSource'
```

[27] Well, large value types, such as big structs, can be expensive to copy—another reason to be aware of what's happening under the hood.

The boxing and unboxing can lead to certain difficulties, particularly because Python doesn't have value types and the programmer will expect everything to behave as a reference.

Take this code that uses Point, which, as a structure, is a value type:

```
>>> from System import Array
>>> from System.Drawing import Point
>>> point = Point(0, 0)
>>> array = Array[Point]((point,))
>>> array[0].X = 30
>>> array[0].X
0
```

This code creates a new Point object (point) and stores it in an array (array). When you fetch the point by indexing the array, you end up setting the X member on an unboxed value—probably not on the object you expected. This problem isn't restricted to Python; if you re-create the code in C#, the same thing happens.

The situation is worse if you attempt to modify a value type exposed as a field (attribute, in Python speak) on an object. Referencing the value type will return a copy, and the update would be lost. In these cases, the IronPython designers have decided[28] that the best thing to do is to raise a ValueError rather than letting the unexpected behavior slip through. As a result, value types are *mostly immutable*,[29] although updates will still be possible by methods exposed on the value types themselves.

8.4.5 Interfaces

Interfaces are a way of specifying that certain types have known behavior. An interface defines functionality, and the methods and properties for that functionality.

Where a class implements an interface, that interface will be added to the *method resolution order*[30] when you use it from Python (effectively making the interface behave like a base class—coincidentally you can also implement an interface in a Python class by inheriting from it). Even if an explicitly implemented interface method is marked as private on a class, you'll still be able to call it.

As well as calling the method directly, you can call the method on the interface—passing in the instance as the first argument in the same way you pass the instance (self) as the first argument when calling an unbound method on a class.[31]

The following snippet shows an example of calling BeginInit on a DataGridView, through the ISupportInitialize interface that it implements:

---

[28] This page has the details: http://channel9.msdn.com/wiki/default.aspx/IronPython.ValueTypes.

[29] In fact, the CLR API design guide says *Do not create mutable value types.*

[30] Accessible through the __mro__ attribute on classes. In pure-Python code, it specifies the lookup order for finding methods where there's multiple inheritance. For a comprehensive reference, see http://www.python.org/download/releases/2.3/mro/.

[31] You can also call methods on the base classes of .NET classes, by passing in the instance as the first argument. This can also be useful when a method obscures a method on a base class making it impossible to call directly from the instance.

```
>>> import clr
>>> clr.AddReference('System.Windows.Forms')
>>> from System.Windows.Forms import DataGridView
>>> from System.ComponentModel import ISupportInitialize
>>> grid = DataGridView()
>>> ISupportInitialize.BeginInit(grid)
```

How is this useful? Well, it's possible for a class to implement two interfaces with conflicting member names. This technique allows you to call a specific interface method.

This workaround is fine for methods, but you don't normally pass in arguments to invoke properties. Instead, you can use `GetValue` and `SetValue` on the interface property descriptor.

```
ISomeInterface.PropertyName.GetValue(instance)
ISomeInterface.PropertyName.SetValue(instance, value)
```

NOTE `GetValue` and `SetValue` aren't available only for property descriptors; they're also available for instance fields if you access them directly on the class in IronPython.

Working with interfaces is mostly straightforward. Unfortunately, when it comes to .NET attributes, it's a much sorrier tale.

8.4.6 Attributes

Attributes in the .NET framework are a bit like decorators in Python and annotations in Java. They're metadata applied to objects that provide information to the CLR. They can be applied to classes, methods, properties, and even method parameters or return values (that is, pretty much everywhere).

Attributes are used to provide documentation, specify runtime information, and even specify behavior. The `DllImport` attribute provides access to unmanaged code; and, in Silverlight, you use the `Scriptable` attribute to expose objects to JavaScript. The following code segment uses the `DllImport` attribute to expose functions `user32.dll` as static methods on a class in C#:

```
[DllImport("user32.dll")]
public static extern IntPtr GetDesktopWindow();

[DllImport("user32.dll")]
public static extern IntPtr GetTopWindow(IntPtr hWnd);
```

Because attributes are for the compiler, they're used at compile time—a problem for IronPython. First, there's no syntax in Python that maps to attributes. The IronPython team is keen to avoid introducing syntax to IronPython that isn't backwards compatible with Python.

A worse problem is that IronPython classes aren't true .NET classes. There are various reasons for this, among them that .NET classes aren't garbage collected and that you can swap out the type of Python classes at runtime! The upshot is that you can't apply attributes to Python classes.

The most common solution is to write stub C# classes and then subclass from IronPython. You can even generate and compile these classes at runtime. We use both of

these techniques later in the book. This isn't the end of the story, though—the IronPython team is still keen to find a solution to this problem that will allow you to use attributes from IronPython.

8.4.7 Static compilation of IronPython code

A feature of IronPython 1 that *nearly* didn't make it into IronPython 2 is the ability to compile Python code into binary assemblies. Fortunately, this feature made a comeback in IronPython 2 beta 4.

IronPython works by compiling Python code to assemblies in memory. If you execute a Python script using the IronPython `ipy.exe` executable, then you can dump these assemblies to disk by launching it with the `-X:SaveAssemblies` command-line arguments. They're not executable on their own though; they're useful for debugging, but they make calls into dynamic call sites[32] generated at runtime when running IronPython scripts.

The `clr` module includes a function that can save executable assemblies: `CompileModules` (IronPython 2 only). Its call signature is as follows:

```
clr.CompileModules(assemblyName, *filenames, **kwArgs)
```

This function compiles Python files into assemblies that you can add references to and import from in the normal way with IronPython. This feature allows you to package several Python modules as a single assembly. The following snippet of code takes two modules and outputs a single assembly:

```
import clr
clr.CompileModules("modules.dll", "module.py", "module2.py")
```

Having created this assembly, you can add a reference to it and then import the `module` and `module2` namespaces from it.

```
import clr
clr.AddReference('modules.dll')
import module, module2
```

`CompileModules` also takes a keyword argument, `mainModule`, that allows you to specify the Python file that acts as the entry point for your application. This still outputs to a dll rather than an exe file, but it can be combined with a stub executable to create binary distributions of Python applications. Creating a stub executable can be automated with some fiddly use of the .NET `Reflection.Emit` API, but it's *far* simpler to use the IronPython Pyc[33] sample. Pyc is a command-line tool that creates console or Windows executables from Python files. It comes with good documentation, plus a command-line help switch. The basic usage pattern is as follows:

```
ipy.exe pyc.py /out:program.exe /main:main.py /target:winexe module.py
➥  module2.py
```

---

32 An implementation detail of the Dynamic Language Runtime.
33 Available for download from the CodePlex IronPython site.

In this chapter, we've climbed under the hood of both the Python language and how IronPython interacts with the .NET framework. It's time to summarize and move on to the next chapter.

8.5 Summary

Python uses protocols to provide features, whereas a language like C# uses interfaces. In addition, a lot of the more dynamic language features of Python, such as customizing attribute access, can be done by implementing protocol methods. You'll find that you use some of these magic methods a great deal. Providing equality and comparison methods on classes is something you'll do frequently when writing Python code. Other protocols, including metaclasses, you'll use only rarely. Even if you don't use them directly, understanding the mechanisms at work deepens your understanding of Python.

We've also worked with some of the areas of the .NET framework that don't easily map to Python. IronPython exposes objects as Python objects; but, in certain places, this is a leaky abstraction. You should now have a clearer understanding of what's going on under the hood with IronPython, especially in the distinction between value and reference types. This understanding is vital when the abstraction breaks down and you need to know exactly what's happening beneath the surface. The specific information we've covered is vital for any non-trivial interaction with .NET classes and objects.

In the next chapter, we use some new features added to the .NET framework in .NET 3.0, including the new user interface library, the Windows Presentation Foundation (WPF). Although younger than Windows Forms, WPF is capable of creating attractive and flexible user interfaces in ways not possible with its older cousin.

Part 3

IronPython and advanced .NET

In the first two parts of this book, we explored the Python language, the .NET framework, and how to get the best from both of them. In this section, we look at how IronPython fits into some interesting and exciting areas of the .NET framework. Since IronPython was first announced, the people at Microsoft have released some impressive new technologies including .NET 3.0, Silverlight, and PowerShell. They've also built IronPython support into existing technologies like ASP.NET. These are interesting topics that give us an opportunity to put IronPython to practical use.

9

WPF and IronPython

This chapter covers

- Introduction to .NET 3.0 and WPF
- WPF and XAML
- Controls and layout
- Loading and transforming XAML at runtime
- XPS documents and flow content

The .NET framework has been around for a while now—the first pre-beta release was in July 2000. You can see the major milestones in the life of the framework in figure 9.1.

The most prevalent installed version of .NET (and the minimum requirement for running IronPython) is version 2.0, which dates back to 2005. It includes dramatic improvements to the C# language (including support for generics), the runtime, and numerous class libraries.

.NET 3.0 and 3.5 build on the 2.0 version of the CLR. Although they add many new assemblies and improvements, they don't include a new version of the CLR. .NET 3.0 is a standard part of Windows Vista and Windows Server 2008, and is available as an add-on for Windows XP and Windows Server 2003.

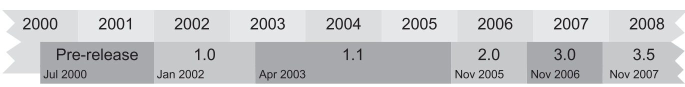


Figure 9.1 Timeline for the .NET framework

NOTE Most of .NET 3.0 hasn't yet been ported to Mono. You can see the status of the Mono implementation, called Olive, at http://www.mono-project.com/Olive.

The four major components in .NET 3.0 are listed in table 9.1.

Table 9.1 The major new APIs of .NET 3.0

Name	Purpose
Windows Presentation Foundation	A new user interface library
Windows Communications Foundation	Library for working with web services
Windows Workflow Foundation	For creating, managing, and working with workflows
Windows CardSpace	For managing digital identities.

These are all large libraries in their own right. We'll be taking a closer look at WPF and using it from IronPython.

WPF (the library formerly known as Avalon) is a new user interface library. Windows Forms, although mature and widely used, is starting to show its age in some areas; it's built on top of User32, for creating the standard interface components, and GDI/GDI+, for rendering shapes, text, and images.

WPF is an entirely new library built on DirectX. DirectX started life as a games technology and is able to use the hardware acceleration provided by modern graphics cards. WPF is not only a more modern and flexible toolkit; but, by offloading rendering to the GPU on the video card, it can also be massively richer without bogging down your processor. WPF includes a standard set of controls and enables the integration of powerful effects such as transparency, vector graphics, animation, and 3D content. Figure 9.2 shows an attractive user interface created with WPF.

Because WPF handles all the drawing of controls, you're free to implement entirely new controls and apply transformations with the minimum of code. To do the same with Windows Forms, you'd have to paint every aspect of the control yourself.

In this chapter, we use WPF from IronPython, exploring some of the new controls it offers. WPF is a large library, going wider than just user interface design. As well as using controls, we work with some of the other features that WPF provides, such as document support. WPF can be used entirely from code, but UIs can also be created from an XML dialect called the eXtensible Application Markup Language, or XAML. We work with WPF from both code and XAML.

XAML allows interface design to be decoupled from application implementation. Interfaces can be created by designers with tools like Expression Blend or the Mono alternative, Lunar Eclipse. These interfaces can then be delivered to developers in the form of XAML for use directly in projects.

Developers may be more comfortable with producing the basic elements of an application from code, but complex elements like gradients and animations are easier to describe with XAML.

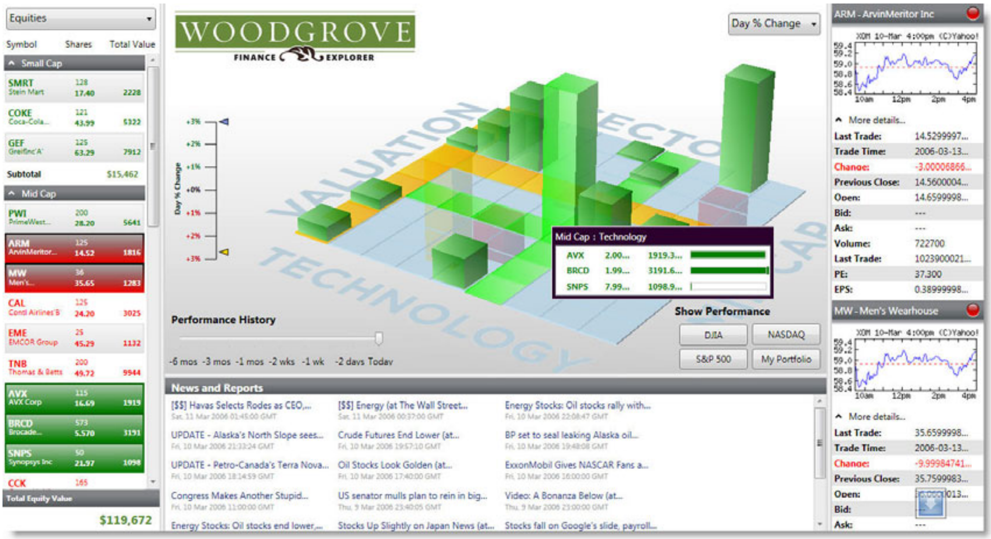


Figure 9.2 Woodgrove Finance—a WPF application

Designer support for creating WPF user interfaces in IronPython is built into Visual Studio 2008 and the IronPython Studio. The IronPython Studio WPF designer is shown in figure 9.3.

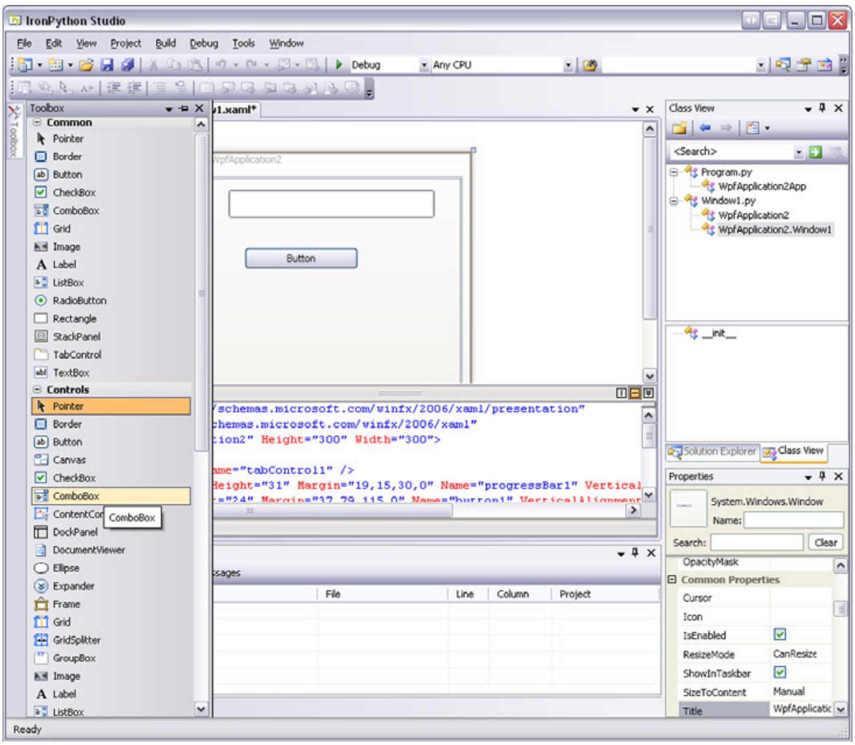


Figure 9.3 The WPF designer in IronPython Studio

Windows Forms is a great user interface library. This may seem like an odd way to introduce an *alternative* library, but bear with us. Windows Forms has been around for a long time and, as a result, is stable with a rich set of controls. WPF opens up new opportunities for creating attractive and innovative interfaces, but it doesn't yet provide as many controls as WinForms. The one you use should depend on your priorities. The great strength of WPF is ultimately flexibility, but it does have some great tricks up its sleeves even with its standard controls. These tricks include better high-level controls for layout, particularly for interfaces that behave well when resized. This tackles a weakness of Windows Forms—creating forms with a consistent resizable layout is notoriously difficult.

We start with a traditional Hello World example, but first let's look at the assemblies that contain WPF, and the namespaces they make available.

9.1 Hello World with WPF and IronPython

To use WPF, you'll need to install .NET 3.0. If you're running Vista, then you already have .NET 3.0 installed.

Installing .NET 3 installs the following key WPF assemblies:

- *PresentationFramework.dll*—Holds the high-level WPF controls and types, such as windows and panels.
- *PresentationCore.dll*—Holds base types from which all the controls and display elements derive, including the core classes `UIElement` and `Visual`.
- *WindowsBase.dll*—Contains lower-level objects that still need to be publicly exposed, such as `DispatcherObject` and `DependencyObject`. We use one or two classes, like `Point`, provided by this assembly.
- *milcore.dll*—The WPF rendering engine (and an essential part of Windows Vista).
- *WindowsCodecs.dll*—A low-level imaging support library.

We can get a long way using only the first two of these assemblies, `PresentationFramework` and `PresentationCore`. These assemblies contain the namespace used by WPF, all of which start with `System.Windows`.[1] (`System.Windows.Forms` is the only namespace in this hierarchy which *isn't* part of WPF.) Table 9.2 shows the important WPF namespaces, all of which we use in this chapter.

Table 9.2 Important WPF namespaces

Namespace	Purpose
`System.Windows`	Provides important classes and base element classes used in WPF applications. This includes `Application`, `Clipboard`, `Window`, and `UIElement`.
`System.Windows.Controls`	Provides the standard WPF controls, plus the classes used to create user controls.

[1] The MSDN documentation for this namespace is at http://msdn2.microsoft.com/en-us/library/system.windows.aspx.

Table 9.2 Important WPF namespaces *(continued)*

Namespace	Purpose
`System.Windows.Documents`	Types for supporting `FlowDocument` and XPS documents.
`System.Windows.Media`	For integrating rich media, including drawings, text, and audio and video content in WPF applications.
`System.Windows.Media.Effects`	Types for applying visual effects to bitmaps.
`System.Windows.Media.Imaging`	Provides classes for working with images (bitmaps).
`System.Windows.Shapes`	Various shapes (such as ellipse, line, path, and polygon, plus the base class `Shape`)—for use in XAML and code.
`System.Windows.Markup`	Classes for working with XAML, including serialization and extensions to XAML.

There's an enormous variety of classes in WPF, but the basics are easy to start with. We begin with a traditional Hello World example.

9.1.1 WPF from code

We've talked enough about WPF—it's time to see it in action. Listing 9.1[2] creates a `Window`, and an attractive-looking button, contained in a `StackPanel`. Figure 9.4 is the result of running the code.



Figure 9.4 Hello World with WPF and IronPython

Listing 9.1 Hello World with WPF and IronPython

```
import clr
clr.AddReference("PresentationFramework")
clr.AddReference("PresentationCore")

from System.Windows import (
  Application, SizeToContent,
  Thickness, Window
)

from System.Windows.Controls import (
  Button, Label, StackPanel
)

from System.Windows.Media.Effects import DropShadowBitmapEffect

window = Window()
window.Title = 'Welcome to IronPython'
window.SizeToContent = SizeToContent.Height
window.Width = 450
```

Adds references to WPF assemblies (annotation on the `clr.AddReference` lines)

Window will update height to content (annotation on `window.SizeToContent = SizeToContent.Height`)

2 Adapted from code originally created by Steve Gilham. See http://stevegilham.blogspot.com/2007/07/hello-wpf-in-ironpython.html.

```
stack = StackPanel()
stack.Margin = Thickness(15)
window.Content = stack

button = Button()
button.Content = 'Push Me'
button.FontSize = 24
button.BitmapEffect = DropShadowBitmapEffect()

def onClick(sender, event):
  message = Label()
  message.FontSize = 36
  message.Content = 'Welcome to IronPython!'

  stack.Children.Add(message)

button.Click += onClick
stack.Children.Add(button)

app = Application()
app.Run(window)
```

Fancy bitmap effect for button

Most of this listing is simple enough that it should mainly speak for itself. One thing you'll notice straightaway is that the API is different than the one in Windows Forms. This is hardly surprising because WPF is an entirely new library; but, although some patterns are familiar, you need to learn a lot of new details.

One obvious difference is at the end of the code. Instead of starting the event loop with a static method on `Application` class, you create an instance. Because you can't have multiple event loops within the same `AppDomain`, `Application` is a singleton. Instantiating it multiple times is not such a hot idea, but it will always return the same instance. Like Windows Forms, WPF has single thread affinity, so interaction with interface components must be on the same thread that `Application` is instantiated on—which must be a single-threaded apartment (STA) thread. For single-threaded applications, you don't need to worry about this; for multithreaded applications, you can use the WPF Dispatcher.[3]

The WPF `Button` is also similar to its WinForms equivalent. You set up the button click handler using our old friend, the `Click` event. But instead of setting the `Text` property, you use `Content`. For setting the text on a button, you use a string, but you *could* host any element in it (like an image or a `TextBlock` for more control over text wrapping and providing an access key).

Setting the `Content` on UI elements is one of the new patterns that comes with WPF. This is how you set the contained elements in the `Window` (the new and improved `Form` of a bygone era).

Setting `Content` isn't the only way that contained elements are set, though. The star of this example is the `StackPanel`, one of the new layout classes provided by WPF. Child controls are set on the stack panel by adding them to the `Children` container. The `StackPanel` is a great way of stacking elements vertically or horizontally. In this

[3] For more details, see the documentation on the WPF threading model at http://msdn2.microsoft.com/en-us/library/ms741870.aspx.

example, you add a new message to the panel every time the button is clicked. Other WPF container classes[4] include `Grid`, `DockPanel`, `WrapPanel`, and the `Canvas`.

Most of the online examples for WPF start by showing you XAML, so let's see how our Hello World example looks in XAML.

9.1.2 Hello World from XAML

XAML is a declarative way of representing user interfaces in XML. XAML itself is extensible; following are several XAML variants:

- *WPF XAML*—Used to describe WPF content
- *XPS XAML*—A subset of WPF XAML for creating formatted electronic documents
- *Silverlight XAML*—Another subset of WPF XAML supported by the Silverlight browser plugin
- *WF XAML*—Windows Workflow Foundation XAML
- *Binary Application Markup Language* (*BAML*)—A compiled form of XAML

There's a straightforward correspondence between XAML and WPF classes. Any XAML can be replaced by code that does the same job.

When XAML is read in, a type converter turns the XML tree into an object tree using the classes, structures, and enumerations contained in the `System.Windows` namespaces. XAML is an abstraction; and, before using an abstraction, we like to understand what's going on under the hood.

You can see the correspondence between the classes we've used and XAML in listing 9.2, which is the XAML for Hello World.

Listing 9.2 Hello World user interface in XAML

```
<Window
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   Title="Welcome to IronPython" Width="450"
   SizeToContent="Height">
  <StackPanel Margin="15" x:Name="stackPanel">
    <Button FontSize="24" x:Name="button">
      <Button.BitmapEffect>
        <DropShadowBitmapEffect />
      </Button.BitmapEffect>
      Push Me
    </Button>
  </StackPanel>
</Window>
```

We'd show you a screenshot, but this is *identical* to the code written in the last section. The top-level element has two XML namespaces, declared as attributes—which is necessary for the text to be valid XML *and* valid XAML.

[4] A useful reference on the WPF layout system can be found at http://msdn2.microsoft.com/en-us/library/ms745058.aspx.

XML escaping rules for XAML

XAML is XML, so you need to escape some characters using XML character entities. Characters that need escaping are in table 9.3.

Character	XML character entity
Less than (<)	`<`
Greater than (>)	`>`
Ampersand (&)	`&`
Quotation mark (")	`"`

Table 9.3 XML character entities

The XML tree defines the layout of the user interface. Child elements are nested in their parent elements.

The attributes of the elements set properties. For example, the `Title`, `Width`, and `SizeToContent` properties on the window are set with attributes on the `Window` tag.

More complex properties (usually objects that have their own properties) are set using *property-element syntax*. You add a child element that specifies the parent and the property being set. In our example, you set the `BitmapEffect` property of the button with the following:

```
<Button.BitmapEffect>
  <DropShadowBitmapEffect />
</Button.BitmapEffect>
```

If you were creating an interface with C# and a WPF designer, then you could also hook up events in the XAML. This creates a partial class in compiled XAML (BAML) that will be associated with a C# class: the code-behind. Because of the differences between IronPython classes and classes created with C#, you can't hook up IronPython event handlers in IronPython.[5] Instead, you have to load the XAML and turn it into a WPF object tree using the `XamlReader`, so you need programmatic access to the elements in the object tree.

In this XAML example, you get access to the `Button` and the `StackPanel` by giving them a name using the `x:Name` attribute. They're then retrieved from the object tree returned by `XamlReader.Load` using the `FindName` method. Listing 9.3 is the code that reads in the XAML and sets up the button event handler.

Listing 9.3 Consuming XAML from IronPython

```
import clr
clr.AddReference("PresentationFramework")
clr.AddReference("PresentationCore")
```

[5] In fact, you *can* do it with the WPF designer in IronPython Studio. This generates code that uses a technique similar to listing 9.3.

```
from System.IO import File
from System.Windows.Markup import XamlReader

from System.Windows import (
  Application, Window
)
from System.Windows.Controls import Label

class HelloWorld(object):
  def __init__(self):
    stream = File.OpenRead("HelloWorld.xaml")
    self.Root = XamlReader.Load(stream)            <-- Creates WPF object from XAML
    self.button = self.Root.FindName('button')     <-- Retrieves button from object tree
    self.stackPanel = self.Root.FindName('stackPanel')
    self.button.Click += self.onClick              <-- Sets click handler

  def onClick(self, sender, event):
    message = Label()
    message.FontSize = 36
    message.Content = 'Welcome to IronPython!'
    self.stackPanel.Children.Add(message)

hello = HelloWorld()

app = Application()
app.Run(hello.Root)
```

Because the XAML creates a `Window`, our `HelloWorld` class is no longer a window subclass itself. In fact, there's no reason that the top-level element needs to be a window. You could make the `StackPanel` the top-level element and set the return value from `XamlReader.Load` as the `Content` on a `Window` subclass.

Given that you're probably a programmer, why should you use XAML instead of code? One reason for considering XAML is for dynamic user interface creation. By using the `XamlReader`, you can read in interface definitions at runtime. Alternative layouts could be loaded in and swapped out in response to user actions, or even created dynamically through text manipulation.

There are other reasons for preferring XAML, as the following fragment illustrates:

```
<LinearGradientBrush>
 <LinearGradientBrush.GradientStops>
  <GradientStop Offset="0.00" Color="Yellow" />
  <GradientStop Offset="0.25" Color="Tomato" />
  <GradientStop Offset="0.75" Color="DeepSkyBlue" />
  <GradientStop Offset="1.00" Color="LimeGreen" />
 </LinearGradientBrush.GradientStops>
</LinearGradientBrush>
```

This XAML fragment creates a colored gradient brush used to create colorful backgrounds. The XAML is less verbose than the equivalent in code, *even more so* if you factor in the imports (and the time it takes to find the right namespaces to import everything from). More importantly, although XAML is quite easy to read and to write by hand, this kind of element is much easier to create with tools like Expression Blend.

Expression Blend allows you (or your designers) to create extremely rich interfaces with transitions, gradients, and animations. If Expression Blend is a designer's

tool, why is it of interest to developers? As well as designing full interfaces, you can also design individual components. You can copy and paste XAML from an Expression project into your code, and vice versa.

XAML generated by Expression Blend, or by the Visual Studio WPF designer, is intended to be compiled and used with code-behind. It will include an `x:Class` attribute valid only in compiled XAML. To make the XAML valid for consumption by the `XamlReader`, you'll need to remove this attribute from the top-level element.

A free trial of Expression Blend is available for download,[6] and Mono has a fledgling equivalent tool (free and open source) called Lunar Eclipse.

As well as creating sophisticated interfaces with Expression, WPF comes with a range of standard controls sufficient for creating attractive, modern-looking applications.

9.2 WPF in action

Although WPF doesn't have as many controls as Windows Forms, it includes standard controls such as check boxes, drop-down lists, and radio buttons. It also includes a range of new controls, both for advanced layout and entirely new user interface components. WPF also covers a wide range of areas beyond traditional user interfaces, including document support and 3D drawing. Even though it doesn't have all the controls that Windows Forms does, it still does an awful lot. It's *extremely* useful for developing Windows applications from IronPython if you're prepared to target .NET 3.0.[7] Although most of the documentation and online tutorials focus on XAML, which often isn't the best way of working with WPF from IronPython, most of the features are as straightforward to use from code as the last example.

In this section, you'll create a WPF application using a selection of controls, both new and old. Although this application itself won't win any design awards, it does show you how to use a useful range of WPF controls. The finished application looks like figure 9.5.

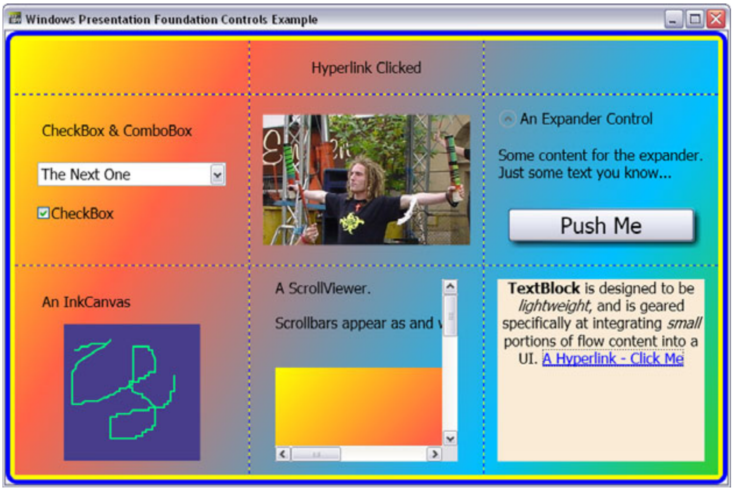


Figure 9.5 A selection of WPF controls in a grid

[6] From Microsoft product page: http://www.microsoft.com/expression/products/overview.aspx?key=blend.

[7] Which will be an extra dependency on platforms older than Vista.

You can see that the controls are laid out in a grid. The code for each of the controls is covered in its own section, starting with the basic framework of the application.

9.2.1 Layout with the Grid

The most important component in this user interface is the `Grid`. This is contained in a border with curved edges, purely for visual effect, and has visible grid lines so that you can see how the controls are contained within it. Listing 9.4 is the constructor for `ControlsExample`[8] and the methods it calls to create the main window. `ControlsExample` itself is a subclass of `Window`.

Listing 9.4 Controls example framework with `Grid` in `Window`

```
import clr
clr.AddReference("PresentationFramework")
clr.AddReference("PresentationCore")
clr.AddReference("windowsbase")

from System.Windows import (
  Window, Thickness,
  HorizontalAlignment,
  SizeToContent, CornerRadius
)

from System.Windows.Controls import (
  Grid, ColumnDefinition, RowDefinition,
  Label, Border
)

from System.Windows.Media import Brushes

class ControlsExample(Window):
  def __init__(self):                        # Sets up grid
    grid = self.getGrid()
    grid.Background = GetLinearGradientBrush()
    self.createControls(grid)                # Creates rest of UI

    border = Border()                        # Creates border
    border.BorderThickness = Thickness(5)
    border.CornerRadius = CornerRadius(10)
    border.BorderBrush = Brushes.Blue
    border.Background = Brushes.Yellow
    border.Padding = Thickness(5)
    border.Child = grid                      # Puts grid in border
    self.Content = border                    # Sets border on window

    self.Title = 'WPF Controls Example'
    self.SizeToContent = SizeToContent.Height
    self.Width = 800

  def getGrid(self):
    grid = Grid()
    grid.ShowGridLines = True
```

[8] This listing is not the *full* code. The following section works through more of the code for `ControlsExample`. You can download it as a single file from the book's website.

```
    # 3x3 grid
    for i in range(3):
      grid.ColumnDefinitions.Add(ColumnDefinition())
      grid.RowDefinitions.Add(RowDefinition())

    label = Label()
    label.Margin = Thickness(15)
    label.FontSize = 16
    label.Content = "Nothing Yet..."
    label.HorizontalAlignment = HorizontalAlignment.Center
    self.label = label

    grid.SetColumnSpan(self.label, 3)
    grid.SetRow(self.label, 0)
    grid.Children.Add(self.label)

    return grid
```

Annotations: "Creates 3 rows and 3 columns" (on the for loop); "Label spans 3 columns" (on grid.SetColumnSpan)

This code initializes the grid and then calls down to `createControls` to create the rest of the controls. Note that the grid is contained in the border by setting the `Child` attribute of the border. Along the way, this code uses a helper function to set a colorful background gradient and another function to place the controls in the grid. These functions, along with the additional imports that they use, are shown in listing 9.5.

Listing 9.5 Helper functions for the controls example

```
from System.Windows import Point
from System.Windows.Controls import ToolTip
from System.Windows.Input import Cursors
from System.Windows.Media import (
  Colors, GradientStop,
  LinearGradientBrush
)

def GetLinearGradientBrush():
  brush = LinearGradientBrush()
  brush.StartPoint = Point(0,0)
  brush.EndPoint = Point(1,1)
  stops = [
    (Colors.Yellow, 0.0),
    (Colors.Tomato, 0.25),
    (Colors.DeepSkyBlue, 0.75),
    (Colors.LimeGreen, 1.0)
  ]
  for color, stop in stops:
    brush.GradientStops.Add(GradientStop(color, stop))
  return brush

def SetGridChild(grid, child, col, row, tooltip):
  if hasattr(child, 'FontSize'):
    child.FontSize = 16
  child.Margin = Thickness(15)
  child.Cursor = Cursors.Hand
  child.ToolTip = ToolTip(Content=tooltip)
  grid.SetColumn(child, col)
  grid.SetRow(child, row)
  grid.Children.Add(child)
```

Annotations: "Creates brush" (on brush = LinearGradientBrush()); "Adds gradient stops" (on brush.GradientStops.Add); "Not all elements have FontSize attributes" (on if hasattr)

`GetLinearGradientBrush` returns a brush (a `LinearGradientBrush`), which is set as the `Background` property on the grid. The start and end points of the gradient are set using the `Point` structure. This isn't the `Point` from the `System.Drawing` namespace that we've used before, but a new one. (In fact, if you attempt to use that one, you get the wonderful error message *expected Point, got Point.*) Although this `Point` lives in the `System.Windows` namespace, it's defined in the `WindowsBase` assembly. This structure is the only reason you need to explicitly add a reference to this assembly at the start of the application.

`SetGridChild` is used to set the controls in the grid. Let's look at how the grid is used.

The grid in this application is three-by-three: three rows and three columns. The rows and columns are created by adding `RowDefinition` and `ColumnDefinition` to their respective collections on the grid object.

```
grid = Grid()
grid.ShowGridLines = True

for i in range(3):
   grid.ColumnDefinitions.Add(ColumnDefinition())
   grid.RowDefinitions.Add(RowDefinition())
```

The first row (row zero) contains the top label, spanning across all three columns. The label has to be set in position in the grid *and* added to the `Children` collection on the grid.

```
grid.SetColumnSpan(self.label, 3)
grid.SetRow(self.label, 0)
grid.Children.Add(self.label)
```

Later controls are added to the grid by `SetGridChild`.

```
grid.SetColumn(child, col)
grid.SetRow(child, row)
grid.Children.Add(child)
```

`SetGridChild` also does a couple of other neat things. It sets a cursor and a tooltip on all the objects it places in the grid. If you move the mouse pointer over the controls, then the mouse pointer becomes a hand and a tooltip for the control is shown.

It's time to look at some of the controls used in this application, starting with a couple of standard controls available in Windows Forms, but have a new implementation for WPF.

9.2.2 The WPF ComboBox and CheckBox

Figure 9.6 shows the WPF `CheckBox` and `ComboBox`.

They're contained in a `StackPanel` and created by the `createComboAndCheck` method (listing 9.6).



Figure 9.6 `CheckBox` **and** `ComboBox`

Listing 9.6 Creating `ComboBox` and `CheckBox`

```
from System.Windows.Controls import (
   StackPanel, CheckBox, ComboBox,
   ComboBoxItem
```

```
)

def createComboAndCheck(self, grid):
  panel = StackPanel()

  label = Label()
  label.Content = "CheckBox & ComboBox"
  label.FontSize = 16
  label.Margin = Thickness(10)

  check = CheckBox()
  check.Content = "CheckBox"
  check.Margin = Thickness(10)
  check.FontSize = 16
  check.IsChecked = True
  def action(s, e):                         <- Called when CheckBox is used
    checked = check.IsChecked
    self.label.Content = "CheckBox IsChecked = %s" % checked
  check.Checked += action
  check.Unchecked += action

  combo = ComboBox()
  for entry in ("A ComboBox", "An Item", "The Next One", "Another"):
    item = ComboBoxItem()                   <- Populates ComboBox
    item.Content = entry
    item.FontSize = 16
    combo.Items.Add(item)
  combo.SelectedIndex = 0
  combo.Height = 26
  def action(s, e):
    selected = combo.SelectedIndex
    self.label.Content = "ComboBox SelectedIndex = %s" % selected
  combo.SelectionChanged += action
  combo.FontSize = 16
  combo.Margin = Thickness(10)

  panel.Children.Add(label)
  panel.Children.Add(combo)
  panel.Children.Add(check)
  SetGridChild(grid, panel, 0, 1, "ComboBox & CheckBox")      <- Puts StackPanel in grid
```

This code is all straightforward. It creates the `CheckBox` and adds an event handler, called `action`, to be called when it's checked or unchecked. When `action` is called, it sets the text on the top label.

Next, the `ComboBox` is created and populated with `ComboBoxItems`. Another `action` event handler is added to the `SelectionChanged` event, and sets the text on the label when the selection is changed.

Finally, these components are placed in the `StackPanel`, which is added to the grid in the first column and second row. A `StackPanel` has no `FontSize` property, so the `FontSize` is set on the individual controls. `SetGridChild` checks for the presence of the `FontSize` property using `hasattr`, and won't attempt to set the `FontSize` on the `StackPanel`.

The `ComboBox` and `CheckBox` are basic components in any user interface. WPF also includes other standard controls such as the `RadioButton`, `ListBox`, `TabControl`, `TextBox`, `RichTextBox`, `TreeView`, `Slider`, `ToolBar`, and `ProgressBar`. Although the

API is different than the one in Windows Forms, you can see that the controls are just as easy to use. The best way to start using them is to experiment and read the documentation that provides simple examples. All these controls live in the `System.Windows.Controls` namespace, and you can find the documentation at http://msdn2.microsoft.com/en-us/library/system.windows.controls.aspx.

Another old friend is the `Image` control, which also has its place in the WPF library.

9.2.3 The Image control

One of the reasons we've chosen to show the `Image` control is to show that, although everything may be shiny and new, it isn't without warts. The first wart is that `Image` is awkward to use from code. This is largely mitigated by creating the image from XAML, but it doesn't help with the second problem: how you specify the location of the image to be shown. In fact, it makes this problem worse. Before we discuss this, let's look at the code (listing 9.7).

Listing 9.7 `Image` control

```
import os
from System import Uri, UriKind
from System.Windows.Controls import Image
from System.Windows.Media.Imaging import BitmapImage

def createImage(self, grid):
  image = Image()                    ❶
  bi = BitmapImage()
  bi.BeginInit()
  image_uri = os.path.join(os.path.dirname(__file__), 'image.jpg')    ❷
  bi.UriSource = Uri(image_uri, UriKind.RelativeOrAbsolute)    ❸
  bi.EndInit()
  image.Source = bi    ❹
  SetGridChild(grid, image, 1, 1, "Image")
```

You can see that this code is verbose. You have to create *both* a `BitmapImage` and an `Image` instance ❶. You specify the location of the image file using a `Uri` ❸, which can be done *only* inside a `BeginInit`/`EndInit` block.[9] The `BitmapImage` is set as the `Source` on the `Image` instance ❹.

It's in specifying the location of the image that the real fun begins. You specify an absolute location on the filesystem ❷. This is fine, if a little ugly, from code because you can construct it dynamically. From XAML, it's impossible—unless your application is always going to run from the same location or you dynamically insert the location into the XAML.

One possible alternative is to use the pack URI syntax.[10] It's slightly odd, but easy enough to use.

```
pack://siteoforigin:,,,/directory_name/image.jpg
```

[9] For more detail on the use of images with WPF, this page is a helpful reference: http://msdn2.microsoft.com/en-us/library/ms748873.aspx.

[10] See this page for all the gory details: http://msdn2.microsoft.com/en-us/library/aa970069.aspx.

Unfortunately, this specifies a location relative to the *executing assembly of the current application*. If you're running the script with IronPython, this means relative to the location of ipy.exe. But creating a custom executable for your own applications is simple, and we explore this topic in chapter 15. If your XAML documents and resources are distributed with your application, then this solution would work. For reference, the equivalent XAML for the image is the following:

```
<Image Source="pack://siteoforigin:,,,/directory_name/image.jpg" />
```

Which is a lot simpler than the code in listing 9.7. Later in this chapter, we look at how to use XAML documents from WPF with a high-level document reading control, including fetching images from arbitrary locations.

After the annoying complexity of working with images, let's take a look at one of the new controls, the `Expander`.

9.2.4 The Expander

The `Expander` is one of our favorite controls. It can contain other controls, which the user can expand or hide. One use for the `Expander` is to provide menus as sidebars, as shown in figure 9.7.

The `Expander` is easy to use; the code from our example application is in listing 9.8.

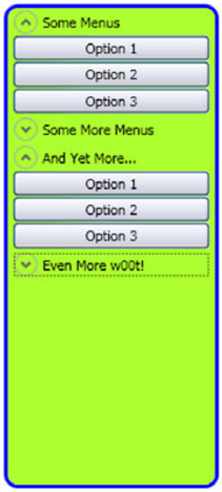


Figure 9.7 `Expander` controls with contained buttons

Listing 9.8 `Expander` control

```
from System.Windows.Controls import (
   Expander, TextBlock, Button
)
from System.Windows.Media.Effects import (
  DropShadowBitmapEffect
)

def createExpander(self, grid):
  expander = Expander()
  expander.Header = "An Expander Control"     <-- Sets title
  expander.IsExpanded = True
  contents = StackPanel()
  textblock = TextBlock(FontSize=16,
            Text="\r\nSome content for the expander..."
                "\r\nJust some text you know...\r\n")
  contents.Children.Add(textblock)
  button = Button()
  button.Content = 'Push Me'
  button.FontSize = 24
  button.BitmapEffect = DropShadowBitmapEffect()
```

```
button.Margin = Thickness(10)
def action(s, e):
  self.label.Content = "Button in Expander pressed"
button.Click += action                                  ◁ Event handler for button
contents.Children.Add(button)

expander.Content = contents                             ◁ Sets StackPanel on Expander
SetGridChild(grid, expander, 2, 1, "Expander")
```

The `Expander` is created and populated with a `StackPanel`, which contains a `TextBlock` and a `Button`. We'll take a closer look at the `TextBlock` in a moment. The `Button` has a `Click` event handler that changes the label, just to prove that it works. Clicking the header hides and shows the contents of the `Expander`.

NOTE The source code for this application includes another control, which we don't have space to cover here. The `InkCanvas` is a control that can be drawn on. You can use it to add user annotations, or diagramming capabilities to an application. It's particularly useful on tablet PCs and can be used in conjunction with the `InkAnalyzer`,[11] which provides handwriting recognition.[12]

Another new WPF component is the `ScrollViewer`.

9.2.5 The ScrollViewer

The `ScrollViewer` is a container element. When the children it contains are bigger than the visible area, it provides horizontal and vertical scrollbars as necessary. Figure 9.8 shows it in action.

Listing 9.9 is the code from our example application that uses the `ScrollViewer` to contain a control and one of the WPF basic shapes with a colored gradient fill. (We thought it looked too good to only use once.)

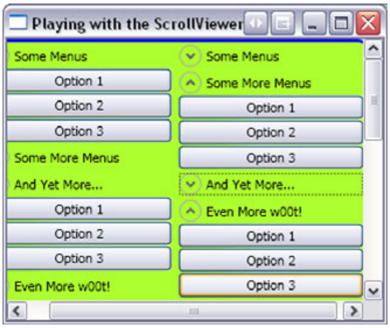


Figure 9.8 The `ScrollViewer` control

Listing 9.9 `ScrollViewer` control with `Rectangle` and `TextBlock`

```
from System.Windows import (
  HorizontalAlignment, VerticalAlignment,
  TextWrapping
)
from System.Windows.Controls import (
   ScrollBarVisibility, ScrollViewer
)
from System.Windows.Shapes import Rectangle

def createScrollViewer(self, grid):
```

[11] See http://msdn2.microsoft.com/en-us/library/system.windows.ink.inkanalyzer.aspx.

[12] To use handwriting recognition, you need to install the tablet PC SDK. We'd include a URL, but it's horrifically long and easy to find.

```
scroll = ScrollViewer()     <— Creates ScrollViewer
scroll.Height = 200
scroll.HorizontalAlignment = HorizontalAlignment.Stretch
scroll.VerticalAlignment = VerticalAlignment.Stretch

scroll.HorizontalScrollBarVisibility = ScrollBarVisibility.Auto
panel = StackPanel()
text = TextBlock()
text.TextWrapping = TextWrapping.Wrap
text.Margin = Thickness(0, 0, 0, 20)
text.Text = "A ScrollViewer.\r\n\r\nScrollbars appear as and when they are
➥ needed...\r\n"

rect = Rectangle()
rect.Fill = GetLinearGradientBrush()
rect.Width = 500
rect.Height = 500

panel.Children.Add(text)
panel.Children.Add(rect)

scroll.Content = panel
SetGridChild(grid, scroll, 1, 2, "ScrollViewer")
```

The `ScrollViewer` contains a `StackPanel` populated with a `TextBlock` and a `Rectangle`. By default, the horizontal scrollbar is disabled, so you enable it by setting `HorizontalScrollBarVisibility` to `ScrollBarVisibility.Auto`. So that the `ScrollViewer` fills the available space, you also set the `HorizontalAlignment` and `VerticalAlignment` to the appropriate enumeration member.

It's also worth noting that this code segment sets the margin on the `TextBlock`, using `Thickness` created with four arguments rather than a single number (technically doubles—but IronPython casts the integers for you).

```
text.Margin = Thickness(0, 0, 0, 20)
```

The four numbers specify the margins on the left, top, right, and bottom. (The `Thickness` structure represents a rectangle.) Here you're specifying a bottom margin of 20 pixels.

The last detail from this code is that the colored gradient is set on the rectangle with the `Fill` attribute. Other brushes are available, like image and tiled drawing brushes,[13] and you can add effects like opacity, reflection, and magnification.

One of the most useful features of WPF is its support for text, in both large and small amounts. The next section covers the `TextBlock`, which is a way of including small amounts of text within your applications.

9.2.6 The TextBlock: a lightweight document control

The `TextBlock` is designed for displaying small amounts of flow content. Flow content is formatted text that will be automatically reflowed as the container control is resized.

[13] As usual, there's a useful page on MSDN providing examples of the different brushes. See http://msdn2.microsoft.com/en-us/library/aa970904.aspx.

As well as the `TextBlock`, there are other controls for incorporating whole documents into applications.

You've already used the `TextBlock` in the `ScrollViewer` example. There you set the content by setting the `Text` property, using it as little more than a glorified label. We used a `TextBlock` rather than a `Label` so that you could control the `TextWrapping`.

In listing 9.10, you'll create a TextBlock with flow content, using the programmatic API.[14] This example uses classes from the `System.Windows.Documents` namespace.

Listing 9.10 `TextBlock` with flow content

```
from System.Windows import TextAlignment
from System.Windows.Documents import (
  Bold, Hyperlink, Italic, Run
)

def createTextBlockAndHyperlink(self, grid):
  textblock = TextBlock()
  textblock.TextWrapping = TextWrapping.Wrap                 <- Sets up attributes on TextBlock
  textblock.Background = Brushes.AntiqueWhite
  textblock.TextAlignment = TextAlignment.Center
  textblock.Inlines.Add(Bold(Run("TextBlock")))              <- Starts adding content
  textblock.Inlines.Add(Run(" is designed to be "))
  textblock.Inlines.Add(Italic(Run("lightweight")))
  textblock.Inlines.Add(Run(", and is geared specifically at integrating "))
  textblock.Inlines.Add(Italic(Run("small")))
  textblock.Inlines.Add(Run(" portions of flow content into a UI. "))

  link = Hyperlink(Run("A Hyperlink - Click Me"))
  def action(s, e):                                          <- Click event handler for Hyperlink
    self.label.Content = "Hyperlink Clicked"
  link.Click += action
  textblock.Inlines.Add(link)
  SetGridChild(grid, textblock, 2, 2, "TextBlock")
```

Documents are another place where XAML is significantly more concise than code. The equivalent XAML for this `TextBlock` is as follows:

```
<TextBlock Background="AntiqueWhite" TextWrapping="Wrap"
   TextAlignment="Center">
   <Bold>TextBlock</Bold> is designed to be
   <Italic>lightweight,</Italic>
   and is geared specifically at integrating
   <Italic>small</Italic> portions of flow content
   into a UI.
   <Hyperlink>A Hyperlink - Click Me</Hyperlink>
</TextBlock>
```

Not only is this less work than the code; but, if you're used to creating documents using HTML (or other markups), it's reasonably intuitive. You'll notice that the code has to wrap straight runs of text in the `Run` class, but this is done automatically in the

[14] For a reference to the `TextBlock` content model, see http://msdn2.microsoft.com/en-us/library/bb613554.aspx.

XAML. The only thing the XAML doesn't do for you is set up the `Click` handler on the hyperlink. This is the same problem that we've already encountered with using XAML from IronPython, and you could solve it using the same trick of setting an `x:Name` attribute in the XAML. Shortly we'll explore a more general solution to this problem when working with XAML documents from IronPython.

Before we do that, let's look at the other side of the coin—turning WPF objects back into XAML.

9.2.7 The XamlWriter

In the Hello World example, we also looked at the equivalent XAML. The full XAML for the example application we've been using to look at WPF controls would be painful to create by hand. Fortunately, there's an easier way. Listing 9.11 uses a `XamlWriter` to turn our `ControlsExample` into XAML.

Listing 9.11 Creating XAML from WPF objects with `XamlWriter`

```
from System.IO import File
from System.Windows.Markup import XamlWriter

window = ControlsExample()
text = XamlWriter.Save(window.Content)
File.WriteAllText('out.xaml', text)
```

The `XamlWriter` does have some limitations. For example, it can't handle creating XAML for subclasses of WPF objects.[15] Our main `ControlsExample` class is a subclass of `Window`, so you can only serialize the object tree below the top-level window. This is why you call `XamlWriter.Save` (a static method) on the window's `Content` property.

We've already looked at including small amounts of flow content in user interfaces. WPF also provides a powerful way of incorporating whole documents through XPS documents.

9.3 XPS documents and flow content

XML Paper Specification (XPS) is a combination of a document markup language, which is a subset of XAML, and code support for viewing and printing documents. Although you won't read this in the Microsoft documentation, many see XPS as Microsoft's answer to Adobe's Portable Document Format (PDF) for creating print-ready documents. Fortunately, that debate is irrelevant to us because the classes that provide XPS support are a *fantastic* way of incorporating documents within WPF applications.

One use for XPS documents is incorporating highly readable content within desktop applications, blurring the line between online and offline applications. One such application is the Mix Reader (figure 9.9) created by Conchango for the Mix UK 2007 conference. It combines offline document reading with online capabilities including RSS reading and Facebook and Twitter integration.

[15] Not just a restriction when working from IronPython. The `XamlWriter` serialization is explicitly runtime and can't access design-time information.



Figure 9.9 The Mix Reader, a desktop WPF application with powerful document-reading capabilities

Applications with dynamic content and features are particularly suited to IronPython. You can dynamically create (and manipulate) XAML and then load it at runtime using the `XamlReader`. In this section, we play with XAML while exploring document integration.

XPS documents are packages that can contain one or more fixed documents along with their dependencies such as images and fonts. This format is most similar to PDF, and Microsoft provides an XPS Viewer application[16] and printing support. In fact, XPS is now the native Windows Vista print spool file format.

Fixed documents are a relatively low-level format that contains `FixedPage` elements. They're used when an application needs to control the layout of the document—for example, for printing—and they're the type of document stored in XPS files. Fixed documents must contain certain elements that make up the page, such as width, height, and language.

More interesting, from an application programmer's point of view, are `FlowDocuments`. Flow documents can be viewed with some high-level reader controls and reflow dynamically as the control size changes. They also offer advanced document features

[16] See http://www.microsoft.com/whdc/xps/viewxps.mspx.

such as pagination and columns. Of course, these features are better demonstrated than explained.

9.3.1 FlowDocument viewer classes

WPF includes several controls for viewing flow content. You've already used the `TextBlock` for including small amounts of formatted text; now it's the turn of the classes for viewing whole documents.

There are three basic classes for viewing flow documents: `FlowDocumentReader`, `FlowDocumentPageViewer`, and `FlowDocumentScrollViewer`. They're all similar, but have slightly different features for different uses.

Figure 9.10 shows FlowDocumentExample.py,[17] which embeds these three document viewers in a `TabControl`. They all show the same document so that you can see the differences.

Like the other WPF controls you've used, you can populate these readers directly from XAML or with an object tree created from code. For documents, it makes much more sense to use XAML, but there's a bit of work to be done from code. Let's look at how you use the viewers and what the differences are.



Figure 9.10 The three flow document viewer classes displaying a flow content document

[17] Available in the downloadable source code, under Chapter 9, from http://www.ironpythoninaction.com/.

THE FLOWDOCUMENTREADER

The `FlowDocumentReader` is the most capable of the viewer controls. It has features that allow the user to switch the display mode between single page, two pages side by side, and continuous scrolling. It also has find and magnification built-in. Listing 9.12 shows the basic code to read in a XAML file and create the viewer.

Listing 9.12 Creating `FlowDocumentReader`

```
from System.IO import File
from System.Windows.Controls import (
   FlowDocumentReader, FlowDocumentReaderViewingMode
)

from System.Windows.Markup import XamlReader

xamlStream = File.OpenRead("FlowDocument.xaml")
reader = FlowDocumentReader()
flowDocument = XamlReader.Load(xamlStream)
reader.Document = flowDocument
reader.ViewingMode = FlowDocumentReaderViewingMode.Page
```

Because the viewers are controls, they live in the `System.Windows.Controls` namespace.

The default viewing mode is `Page`; so, strictly speaking, the last line of the code is unnecessary, but it demonstrates how you configure the viewing mode. The other enumeration members, for the alternative viewing modes, are `Scroll` and `TwoPage`.

THE FLOWDOCUMENTPAGEVIEWER AND FLOWDOCUMENTSCROLLVIEWER

The other two viewer controls are used in exactly the same way, but present the document in a fixed viewing mode instead of allowing the user to choose. In consequence, they're lighter weight and should be used when different viewing modes aren't needed.

The `FlowDocumentPageViewer` shows documents in page-at-a-time mode and includes the magnification toolbar.

The `FlowDocumentScrollViewer` shows documents in single-page view mode, with a vertical scrollbar. A horizontal scrollbar is only shown if necessary. By default, this viewer has no toolbar, but you can switch it on by setting the `IsToolBarVisible` property to `True`.

So what sort of markup do you use to create flow documents?

9.3.2 *Flow document markup*

You've already seen a simple example of the XAML flow document markup when creating the `TextBlock`. But that only scratches the surface of what's possible.

The document displayed by FlowDocumentExample.py is FlowDocument.xaml. This shows off most of the document markup available.

BASIC MARKUP

When creating a viewer control, you need to set the `Document` property with a `FlowDocument`, which must be the top-level element of the document. The basic markup elements of flow content are easy to use. This short document contains text in a paragraph, with bold and italic sections and a line break.

```
<FlowDocument xmlns="… >
   <Paragraph>
      The markup enables me to make some things
      <Bold>Bold</Bold>, and other things
      <Italic>Italic</Italic>. Although it is another
      markup to learn, it is much easier to
      construct documents like this
      than to do it from code.
      <LineBreak />
      This text follows a line break.
   </Paragraph>
</FlowDocument>
```

All these elements have corresponding classes in the `System.Windows.Documents` namespace, but constructing a document from code would be extremely tedious.

As well as for dividing documents into paragraphs, you can use sections as container elements. They can be useful ways of forcing a page break or applying styling attributes to all contained elements. Children of a section must be block-level elements, which include the following:

- `Paragraph`
- `List`
- `Table`
- `BlockUIContainer`

Following are two section declarations. The first ensures a page break; the second adds a light blue background to everything within the section.

```
<Section BreakPageBefore="True"/>
<Section Background="LightBlue"> </Section>
```

Lists are also easy to construct.

```
<List>
   <ListItem>
      <Paragraph><Italic>Item Number 1 - Italic</Italic></Paragraph>
   </ListItem>
   <ListItem>
      <Paragraph><Underline>Item Number 2 - Underline</Underline></Paragraph>
   </ListItem>
</List>
```

Flow content can contain many other elements including figures, typography elements, and embedded user interfaces. You can find examples of these in FlowDocument.xaml. We haven't yet covered images and hyperlinks because both of these elements have problems that we need to solve.

9.3.3 Document XAML and object tree processing

XAML is essentially an XML tree used to represent documents and user interfaces. If you need to apply transformations or changes to the tree, you have two choices. You can either process the XAML before loading, or apply changes directly to the object tree after loading. So far, we've encountered two reasons why you might want to do that. The first is to resolve the difficulty with specifying the location of images.

HANDLING IMAGES

For an image element to be properly displayed, you need to provide an absolute location for the image file. This is inconvenient, and it would be much better to specify the location relative to the document file. We've already solved this problem when creating images from code, but you need to do something different when working with XAML.

Because we're working with an XML tree, you could use an XML parser or apply XSLT transformations. But for specifying a location on the filesystem relative to the document, all you need is a placeholder representing the directory of the document. You can then do a string replace to insert that location into the XAML at runtime. Because you haven't yet used regular expressions (regex), we create a simple regex to do the job.

For regular expressions, you have the choice of working with the .NET *or* the Python libraries. Because we're more familiar with Python regular expressions, we use Python's `re` module[18] (listing 9.13)

Listing 9.13 Regular expression to insert image locations into XAML at runtime

```
from os.path import abspath, dirname, join
import re
from System.IO import File

document_location = re.compile(r'<%\sdocument_location\s%>',
                               re.IGNORECASE)

documentDirectory = abspath(dirname(__file__))
filename = join(documentDirectory, "FlowDocument.xaml")
rawXAML = File.ReadAllText(filename)

if not documentDirectory.endswith('\\'):
   documentDirectory += ' \\'

xaml = re.sub(document_location, documentDirectory, rawXAML)
```

Annotations: **Creates compiled regular expression object** (document_location = re.compile(...)); **Directory containing document** (documentDirectory = abspath(...)); **Performs substitution** (xaml = re.sub(...)).

This code takes an image tag, `<Image Source="<% document_location %>image2.jpg" />`, and replaces `<% document_location %>` with the path to the directory containing the document. The advantage of using a regular expression is that the replacement can be case insensitive and also insensitive to whitespace inside the tag.

This leaves a further problem. You now have the XAML, with the correct image paths, as a string, but the `XamlReader` expects a stream. You get around this problem by using a `MemoryStream` to wrap the string (listing 9.14).

Listing 9.14 Wrapping a string as a stream

```
from System.IO import MemoryStream
from System.Text import Encoding

bytes = Encoding.UTF8.GetBytes(xaml)
stream = MemoryStream(bytes)

flowDocument = XamlReader.Load(stream)
```

[18] See http://docs.python.org/lib/module-re.html.

XAML files *must* be in the UTF-8 encoding—which is why you use `Encoding.UTF8` to get a byte array from the XAML string.

The next problem we have to solve is working with links in documents.

HANDLING LINKS

Constructing hyperlinks in XAML is easy; but, unfortunately, they don't work quite how you might expect.

```
<Hyperlink NavigateUri="http://example.com">A Link</Hyperlink>
```

The `NavigateUri` isn't intended for linking to arbitrary URLs, but for navigating to XAML files. If these were created with code-behind, then the associated code would be loaded with the XAML, and you could construct `Page`[19]-based user interfaces (for both desktop applications and WPF applications hosted in a browser) that navigate like a web application. Because you can't use code-behind with IronPython, it isn't useful for our purposes, and hyperlinks in our XAML documents don't do anything.

But you can load the XAML and then find all the links in the object tree. You can then attach click handlers to the links that launch the default system browser with the URL specified in the `NavigateUri`.

To do this, you need a way of walking the WPF object tree after you've created it. Fortunately, there's a convenient helper class for doing this. Listing 9.15 shows a recursive generator function that uses `LogicalTreeHelper` to extract a list of objects. You specify the objects you're interested in by name, and `FindChildren` examines the class name of all the objects in the WPF tree to find them. Using a generator means that you don't have to build up a list, but can yield matching objects as you find them.

Listing 9.15 Attaching click handlers to all hyperlinks in a document

```
from System.Windows import LogicalTreeHelper
from System.Diagnostics import Process

def FindChildren(child, name):
  if child.__class__.__name__ == name:
    yield child
  for item in LogicalTreeHelper.GetChildren(child):
    if isinstance(item, basestring):          <- Ignores children that are strings
      continue
    for entry in FindChildren(item, name):
      yield entry                              <- Returns children of object passed in

flowDocument = XamlReader.Load(stream)
def OnClick(sender, event):
  uri = sender.NavigateUri
  Process.Start(uri.AbsoluteUri)               <- Launches browser to hyperlink NavigateUri

for link in FindChildren(tree, 'Hyperlink'):
  link.Click += OnClick
```

It would be simple to evolve this system to automatically hook up user interface elements embedded in documents. You could create a declarative naming scheme and

[19] See http://msdn2.microsoft.com/en-us/library/system.windows.controls.page.aspx.

hook events up to handlers based on names declared in XAML elements. For example, an object named `Click_onClick` would have its `Click` event hooked up to the `onClick` method.

Although this section has focused on using `LogicalTreeHelper` in the context of documents, there's no need to restrict its use to this. XAML is just text, and Python is a particularly good language for text processing; even if you prefer working with code, the possibilities for dynamically generating XAML are interesting (especially for complex effects like animations and transitions).

9.4 Summary

WPF is a huge subject. Whole books have been written on WPF (quite a few of them), so we've only scratched the surface. The aim of this chapter was to familiarize you with the basic principles of working with WPF, and equip you to explore it yourself. If you want to learn more, you can begin with the following topics:

- Dependency properties and routed events
- Data binding
- Creating custom controls
- 3D drawing and animations
- Hosting Windows Forms controls in WPF

We've covered how to create WPF applications from IronPython and how to find your way around the WPF namespaces and standard controls. Perhaps more importantly, at least from the point of view of reading WPF documentation, is understanding XAML. Even if you prefer working with objects from code (which we do), you need to be able to read XAML to look up anything related to WPF! When you do work with XAML, you have lots of options with IronPython. The `XamlReader`, `XamlWriter`, and `LogicalTreeHelper` classes are particularly powerful tools.

In the next chapter, we look at how IronPython can help with system administration tasks, ranging from simple scripts to remote monitoring of system resources. Along the way, you'll get a chance to see how you can use some of the libraries provided with PowerShell from IronPython.

10 Windows system administration with IronPython

This chapter covers

- Shell scripting and Python libraries
- Windows Management Instrumentation from .NET
- Remote administration
- Hosting PowerShell from IronPython
- Hosting IronPython from PowerShell

Because Python is interpreted, it's often referred to as a scripting language. The Python community regards this term as slightly derogatory because it seems to imply that Python is suited to only simple scripting tasks rather than larger applications. Having said that, Python does make a *great* scripting language. Scripts can be kept as text source files, common tasks can be achieved with very little code, and you don't need to use classes or even functions if they aren't appropriate for the job at hand. The greatest advantage of Python for system administration tasks is that, as

a full programming language, it's easy to migrate what starts as a simple script into a full application.

In this chapter, we look at system administration with IronPython, taking advantage of the features that Python and the .NET framework provide. The aspects of .NET that we use are Windows Management Instrumentation (WMI) and PowerShell, both of which are frameworks aimed particularly at the systems administrator.

10.1 System administration with Python

Every computer user does *some* system administration, even if it's only maintaining a stable and working system. System administration encompasses everything from keeping a computer operating to maintaining large networks with many computers and servers. Although these are radically different situations, they share some needs and techniques in common. We start our look at administration with an example of simple scripting.

10.1.1 Simple scripts

For simple tasks, one of Python's great advantages is that it doesn't push any particular programming paradigm. If you want to write a script to automate a regular task, you aren't forced to write an object-oriented application; you aren't even forced to write functions if the task at hand doesn't call for them. Listing 10.1 is a script for a typical admin task of clearing out the temp folder of files that haven't been modified for more than seven days.

Listing 10.1 Script to clear out temp folder

```
import os, stat
from datetime import datetime, timedelta

tempdir = os.environ["TEMP"]
max_age = datetime.now() - timedelta(7)

for filename in os.listdir(tempdir):
   path = os.path.join(tempdir, filename)
   if os.path.isdir(path):
      continue
   date_stamp = os.stat(path).st_mtime
   mtime = datetime.fromtimestamp(date_stamp)
   if mtime < max_age:
      mode = os.stat(path).st_mode
      os.chmod(path, mode | stat.S_IWRITE)
      os.remove(path)
```

Annotations: Python standard library modules (import lines); Temp directory environment variable (tempdir/max_age lines); Makes file writeable (os.chmod line)

Python has a rich tradition of being used for shell scripting, particularly on the Linux platform. Commands are executed on the command line, and output their results on standard out, often as a series of lines. Commands that work on multiple files can often accept input from standard input, so commands can be chained together; the output from one script forms the input to the next.

Microsoft has extended the shell scripting concept with PowerShell; you can pipe objects, as well as text, between commands. We look at integrating IronPython with

PowerShell later in this chapter, but first we use IronPython to create more flexible shell scripts.

10.1.2 Shell scripting with IronPython

Because Python is widely used by system administrators, it has grown many libraries to make their lives easier, both in the standard library and third-party libraries. The script in listing 10.1 uses the Python standard library modules `os`, `stat`, and `datetime` to work with paths, files, and dates. There are many more standard library modules, and table 10.1 lists some particularly useful for scripting.

Table 10.1 Standard library modules useful for shell scripting

Name	Purpose
`os`	Working with processes and operating system–specific information
`os.path`	Handling files and paths
`sys`	Containing more system-specific information and the standard input, output, and error streams
`stat`	Interpreting calls to `os.stat()`
`shutil`	High-level file operations including copying, moving, and deleting trees of directories
`glob`	Pathname pattern expansion
`fnmatch`	Path and filename pattern matching
`filecmp`	File and directory comparison

Two common needs in command-line scripts are to interpret command-line arguments and to read from configuration files. Although the Python standard library *does* include modules for these tasks,[1] alternative libraries do the job better, as we demonstrate in our next example.

Python libraries and IRONPYTHONPATH

To use Python standard library modules from IronPython scripts, you need to install IronPython 2 from the msi installer (that includes the standard library) or ensure that Python is installed (Python 2.4 if you're using IronPython 1, or Python 2.5 for IronPython 2). If you do the latter, you'll also need the standard library pointed to by the `IRONPYTHONPATH` environment variable. On Windows, this will usually be the directory C:\Python24\lib or something similar.

It's also useful to have a directory to keep modules that aren't in the standard library. To have multiple directories in `IRONPYTHONPATH`, they should be separated by semicolons—for example:

```
C:\Python24\lib;C:\Modules
```

[1] `optparse` and `ConfigParser`, respectively.

One command invaluable on UNIX-like systems, but missing from the Windows command-line environment, is the `find` command. Windows does have a command called `find`, but it's for searching for text in files and the search feature of the Explorer user interface isn't a replacement for command-line search. The UNIX `find` does a *massive* range of different things, but the functionality I (Michael) miss most is searching a path for files whose name matches (or doesn't match) particular patterns. Let's see how much Python code it takes to implement this functionality.

The specification for `search.py` is as follows:

- Accepting a path, or list of paths, to search for files (defaulting to the current directory)
- Accepting a pattern to match filenames with, using the standard * and ? wildcards and defaulting to everything
- Accepting a pattern to exclude files
- A mechanism for excluding specific directories from the search
- Printing matching files to standard output on individual lines

As a first step, let's look at how our script can meet that specification by accepting command-line arguments.

PARSING COMMAND-LINE ARGUMENTS

Command-line arguments are exposed to you in their raw form as `sys.argv`, but you can make life easier by using a module called `argparse`[2] written by Steven Bethard. Listing 10.2 is a simple function that creates an argument parser and uses it to parse the arguments passed at the command line.

Listing 10.2 Parsing command-line arguments with `argparse`

```
from argparse import ArgumentParser

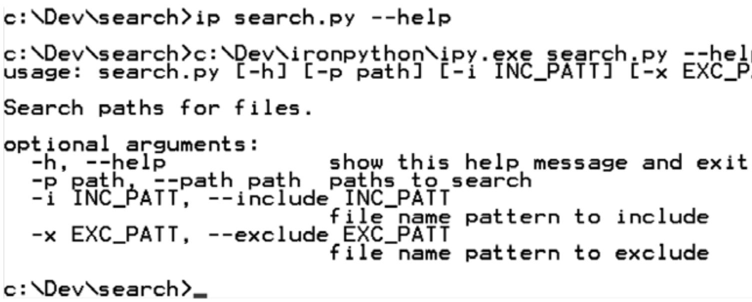
def ParseArgs():
   description = "Search paths for files."
   parser = ArgumentParser(description=description)
   parser.add_argument('-p', '--path',
                  action='append', dest='paths',
                  default=[],
                  metavar='path',
                  help='paths to search',
                 )
   parser.add_argument('-i', '--include',
                  action='store', dest='inc_patt',
                  default='*',
                  help='file name pattern to include',
                 )
   parser.add_argument('-x', '--exclude',
                  action='store', dest='exc_patt',
                  default=None,
                  help='file name pattern to exclude',
                 )
   return parser.parse_args()
```

Annotations: "Printed in help message" (description line); "Default is to match all files" (default='*'); "Default is to exclude none" (default=None).

[2] It isn't yet in the standard library, but will be by Python 3.0. You can find `argparse` at http://argparse.python-hosting.com.

This snippet only uses the most basic features of `argparse`, which has a great deal more functionality that we haven't needed to use. To be able to specify multiple path arguments, you use the `append` action rather than the `store` action for this argument, and provide an empty list as the default value.

You get two of the nicest features of `argparse` with no deliberate effort. If a user calls the script with an invalid set of arguments, `argparse` will print a helpful error message and exit. Additionally, it automatically generates a useful help message if the script is called with the arguments `-h` or `--help`. You can see this message in figure 10.1.



```
c:\Dev\search>ip search.py --help

c:\Dev\search>c:\Dev\ironpython\ipy.exe search.py --help
usage: search.py [-h] [-p path] [-i INC_PATT] [-x EXC_PATT]

Search paths for files.

optional arguments:
  -h, --help            show this help message and exit
  -p path, --path path  paths to search
  -i INC_PATT, --include INC_PATT
                        file name pattern to include
  -x EXC_PATT, --exclude EXC_PATT
                        file name pattern to exclude

c:\Dev\search>_
```

Figure 10.1 The help message generated for search.py by `argparse`

As well as handling command-line arguments, you need a way of specifying directories to exclude from the search. We regularly work with Subversion repositories and need to locate files within them. Subversion repositories on the filesystem keep copies of the working base of files under version control in hidden directories called .svn. Any file that matches will inevitably also match a copy, so we like to be able to exclude all .svn directories from searches. One way of doing this is through a configuration file.

READING ONFIGURATION FILES

I prefer the `ConfigObj`[3] module for reading (and writing) ini-style configuration files. This module also has many advanced options, but makes simple access to config files trivially easy. Listing 10.3 is a function to read a list of excluded directories from a config file called search.ini, which is stored in the user's home directory.

Listing 10.3 Reading config files with `ConfigObj`

```
from configobj import ConfigObj

def GetExcludesFromConfig():
   home = os.path.join(os.getenv('HOMEDRIVE'),
                 os.getenv('HOMEPATH'))

   rcfile = os.path.join(home, 'search.ini')
   config = ConfigObj(rcfile)
   exclude_dirs = config.get('exclude', [])
   if not isinstance(exclude_dirs, list):
      exclude_dirs = [exclude_dirs]

   return exclude_dirs
```

[3] Disclaimer: I am one of the authors of `ConfigObj`. You can find it at http://pypi.python.org/pypi/ConfigObj/.

This code first constructs the path to the user's home directory by combining the environment variables HOMEDRIVE and HOMEPATH. On a UNIX-type system, you could simply use the HOME environment variable. Another alternative would be to call os.expanduser(~), which does the same thing under the hood.

You can access the config file by creating a ConfigObj instance with the path to the file. You don't have to worry about whether this file exists or not. By default ConfigObj doesn't raise an exception if the file doesn't exist because you may be creating a new one.

Normally ini files store key/value pairs, in sections defined by names in square brackets. ConfigObj doesn't require values to be in a section—which is useful for simple configuration files. It will also read a comma-separated list of values into a list of strings for you. The search.ini file only needs to be a text file with a single entry, exclude_dirs.

```
exclude_dirs = '.svn', '.cvs'
```

Having read in the config file, you can access the members using dictionary-like access: exclude_dirs = config['exclude_dirs']. Unfortunately, if the config file isn't found or the exclude_dirs member isn't present, then a KeyError exception is raised. Instead, you can use the get method to fetch the value, supplying a default value of an empty list if exclude_dirs isn't available.

If there was only a single value and the user forgot the trailing comma to make it a list, then exclude_dirs would be read in as a string instead of a list. Before returning the list of excluded directories, you check that it is a list; and, if it isn't, you turn it into one.

You now have the config file and command-line handling written, but the script needs to be able to recursively walk directories returning filenames for you to filter.

WALKING DIRECTORIES

The Python standard library does contain a function for traversing directory trees (os.walk), but it doesn't include a mechanism for easily excluding directories and its interface isn't ideal for our use case. Fortunately, this is an ideal situation for a simple Python generator. Reinventing the wheel may be bad as a general practice; but, if it can be done in ten lines of Python, then it's worth making an exception! Listing 10.4 recursively walks a directory tree, skipping directories in the exclude list, yielding filenames as it finds them.

Listing 10.4 A generator for walking directory trees

```
import os

def walk(directory, exclude_dirs):
   for entry in os.listdir(directory):
      path = os.path.join(directory, entry)
      if os.path.isfile(path):
         yield path                  <-- Yields full paths of files
      elif os.path.isdir(path):
         if entry in exclude_dirs:
            continue

         for member in walk(path, exclude_dirs):     <-- Recurses into subdirectories
            yield member
```

The final piece of the puzzle is to use the helper functions we've written to filter the filenames and print the relevant ones to standard out.

FILTERING FILENAMES

To filter the filenames, using any include or exclude patterns that the user may have supplied, you can use the `fnmatch`[4] module. To do this, you need to *use* the arguments as parsed by `argparse`. Listing 10.5 shows the `search` function, which iterates over all the paths the user has specified, walks them with the `walk` function we just wrote, and filters filenames using `fnmatch.fnmatch`. It also has a section of code that glues everything together and runs when search.py is executed as a script.

Listing 10.5 Filtering filenames using `fnmatch.fnmatch`

```
import fnmatch

def search(exclude_dirs, paths, inc_pattern, exc_pattern):
   if not paths:
      paths = ['.']                        <-- Default if no paths are specified
   for path in paths:                                        <-- Searches all user-specified paths
      for file_path in walk(path, exclude_dirs):
         base_path, filename = os.path.split(file_path)
         if fnmatch.fnmatch(filename, inc_pattern):
            if (not exc_pattern or not
               fnmatch.fnmatch(filename, exc_pattern)):
               print file_path

if __name__ == '__main__':
   exclude_dirs = GetExcludesFromConfig()
   args = ParseArgs()

   search(exclude_dirs, args.paths, args.inc_patt,
        args.exc_patt)
```

One advantage of protecting the execution code with `if __name__ == '__main__'` is that search.py can be imported from as a module as well as executed as a script. The code is reusable; but, more importantly, you could (should!) write unit tests for the individual functions.

You can see in this listing how `ParseArgs` returns the arguments it has parsed. It returns them as a single object, and the individual arguments are accessed using the attribute names specified as the `dest` argument. The `search` function iterates over all the paths returned by `walk` (which handles excluding directories for you), and then filters the paths based on whether they do or do not match the include and exclude patterns.

So far we've accomplished writing a useful, and easily extensible, shell script in seventy lines of Python code.[5] Extending this script—for example, to take an extra command-line argument to return only files newer than a certain file in order to pipe the output to a backup script—would be simple.

---

[4] See http://docs.python.org/lib/module-fnmatch.html.

[5] For useful hints on working with Python scripts from the command line, read the following article: http://www.voidspace.org.uk/python/articles/command_line.shtml.

As well as the flexibility of Python for creating admin tools, you also have the power of .NET at your fingertips. The Windows operating system includes a powerful system, aimed specifically at system administration, called Windows Management Instrumentation.

10.2 WMI and the System.Management assembly

One of the primary Windows interfaces for system management is Management Instrumentation, known affectionately by the acronym WMI. WMI is a management infrastructure, through which system components provide information about their state and notification of events. You can use WMI to change configuration, interrogate the local system or remote computers, and respond to events. Practical uses for WMI include tasks like inventorying all installed software, uninstalling programs, creating scheduled tasks, and obtaining information about running services. Additionally, applications can provide instrumentation so that they can be queried by WMI.

Despite having Windows in the name, WMI is an implementation of the platform-independent Web-Based Enterprise Management (WBEM) and Common Information Model (CIM) standards. But, although parts of the necessary components have been implemented in Mono, large parts of it are considered too Windows-specific and will probably never be implemented. Sadly, this means that most of the examples in this section don't work with Mono.

Although WMI provides you with access to some very low-level system information, it has a good high-level managed interface, in the form of the `System.Management` namespace. This makes it easier to work with WMI through .NET and IronPython than some of the alternatives.

10.2.1 System.Management

`System.Management` provides a managed interface to the WMI infrastructure. The core classes are `ManagementObjectSearcher`, `ManagementQuery`, and `Management-Eventwatcher`. WMI queries are created using Windows Query Language (WQL), which is a derivative of SQL. Much of working with WMI involves knowing how to construct your WQL queries.[6]

SIMPLE WQL QUERIES

Listing 10.6 shows a basic example of WMI that queries and prints the processor usage percentage every five seconds.

Listing 10.6 A simple WQL query to display CPU usage

```
import clr
clr.AddReference("System.Management")
from System.Management  import ManagementObject
from System.Threading import Thread

query = "Win32_PerfFormattedData_PerfOS_Processor.Name='_total'"
```

WQL query for processor usage (annotation on the `query` line)

[6] The Microsoft reference is at http://msdn2.microsoft.com/en-us/library/aa394606.aspx.

```
while True:
   mo = ManagementObject(query)
   print mo["PercentProcessorTime"]
   Thread.CurrentThread.Join(5000)
```

Loop, checking every five seconds

ManagementObjectSearcher is a more commonly used way of executing queries, and it will return a collection of management objects. For example, listing 10.7 queries the system for information about all the attached logical disks.

Listing 10.7 Querying the system with ManagementObjectSearcher

```
import clr
clr.AddReference("System.Management")
from System.Management import ManagementObjectSearcher

query = "Select * from Win32_LogicalDisk"
searcher = ManagementObjectSearcher(query)

for drive in searcher.Get():
   for p in drive.Properties:
      print p.Name, p.Value
   print
```

If you know the property that you're interested in, you can index instead of going through drive.Properties. For example, to get the drive name you can use drive["Name"].

MONITORING EVENTS

Things get interesting when you start to monitor events. For this, you use the ManagementEventWatcher class. Listing 10.8 creates a watcher that calls an event handler when new processes start.

Listing 10.8 Responding to events with ManagementEventWatcher

```
import clr
clr.AddReference('System.Management')
from System.Management import (
   WqlEventQuery, ManagementEventWatcher
)
from System import TimeSpan
from System.Threading import Thread

timeout = TimeSpan(0, 0, 1)
query = WqlEventQuery("__InstanceCreationEvent", timeout,
                 'TargetInstance isa "Win32_Process"')

watcher = ManagementEventWatcher()
watcher.Query = query

def arrived(sender, event):
   print 'Event arrived'
   real_event = event.NewEvent
   instance = real_event['TargetInstance']

   for entry in instance.Properties:
```

Fetches real event!

Fetches process instance

```
        print entry.Name, entry.Value

watcher.EventArrived += arrived
watcher.Start()

while True:                                    <-- Waits for events
    Thread.CurrentThread.Join(1000)
```

This code is really very simple. All the magic happens in constructing the WQL query and adding an event handler to the event watcher instance. Under the hood, `WqlEventQuery` constructs the following WQL query:

```
select * from __InstanceCreationEvent
within 1 where TargetInstance isa "Win32_Process"
```

You specify a timeout when you construct the query (using `System.Timespan`); the timeout corresponds to the `within` clause of the WQL query. Some events have a built-in mechanism for notifying WMI (WMI event providers); these are called *extrinsic events*. WMI discovers other events, *intrinsic events*, by polling, and the timeout tells WMI how often to poll for you.

This code snippet listens for events by hooking up a handler to the `EventArrived` event. Instead of using this event, you can make a call to `watcher.WaitForNextEvent`, which blocks until the event is raised. In this situation, you can also set a timeout directly on the watcher. Instead of blocking forever, the timeout causes the watcher to throw an exception if an event isn't raised in time. The following snippet shows this in practice:

```
>>> watcher = ManagementEventWatcher()
>>> watcher.Query = query
>>> watcher.Options.Timeout = TimeSpan(0, 0, 5)
>>> e = watcher.WaitForNextEvent()
Traceback (most recent call last):
SystemError: Timed out
```

As we mentioned, the secret knowledge needed for harnessing WMI is how to construct your WQL queries. For example, to be notified of new USB storage devices becoming available (plug-and-play events), you could use this query:

```
wql = ("Targetinstance isa 'Win32_PNPEntity' and "
       "TargetInstance.DeviceId like '%USBStor%'")
query = WqlEventQuery("__InstanceCreationEvent", timeout, wql)
```

Let's look a bit more at WQL and the elements available to you to construct queries.

WQL, WMI CLASSES, AND EVENTS

The basic pattern for WQL notification queries is as follows:

```
SELECT * FROM __EventClass WITHIN PollingInterval WHERE TargetInstance ISA
      WMIClassName AND TargetInstance.WMIClassPropertyName = Value
```

The key to constructing useful queries is knowing which events, classes, and properties provide you with the information you need.

Intrinsic events are represented by classes derived from one of the following:

- __InstanceOperationEvent
- __NamespaceOperationEvent
- __ClassOperationEvent

The instance events, which are the most common, are as follow:

- __InstanceCreationEvent
- __InstanceModificationEvent
- __InstanceDeletionEvent

Extrinsic events derive from the `__ExtrinsicEvent` class.

When an event is raised, the corresponding WMI class is instantiated; this is the `TargetInstance` we've already used in some of our examples. You can navigate the documentation for all the standard WMI classes at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/wmisdk/wmi/wmi_classes.asp.

Once you have an event, and have pulled the target instance out, you can explore the interesting properties through the `Properties` collection. Various tools are available to investigate WMI namespaces and all the classes they provide.[7]

Sometimes it's useful to work *directly* with these classes—which you do by creating an instance of `ManagementClass` corresponding to the WMI class you're interested in. Listing 10.9 illustrates this by creating events with a timer.

Listing 10.9 Creating timer events with `ManagementClass`

```
from System.Management import ManagementClass, WqlEventQuery

TimerClass = ManagementClass("__IntervalTimerInstruction")
timer = TimerClass.CreateInstance()
timer["TimerId"] = "Timer1"
timer["IntervalBetweenEvents"] = 1000
timer.Put()                                      <- Starts timer

query = WqlEventQuery("__TimerEvent", "TimerId='Timer1'")   <- Query for timer events
```

This code[8] would be useful for making your WMI demos a bit more predictable; but beyond that, it doesn't have much practical application. Fortunately, you can do more useful things with `ManagementClass`, such as listing all the processes that run on startup.

```
>>> StartupClass = ManagementClass('Win32_StartupCommand')
>>> processes = StartupClass.GetInstances()
>>> for p in processes:
...     print p['Location'], p['Caption'], p['Command']
```

[7] For example, Marc, The PowerShell Guy, has one tool aimed at PowerShell but useful for anyone interested in WMI. See http://thepowershellguy.com/blogs/posh/archive/2007/03/22/powershell-wmi-explorer-part-1.aspx.

[8] The `timer.Put()` line of this example requires administrator access under Vista.

As well as interesting properties, many WMI instances also have useful methods (although not `Win32_StartupCommand`, as it happens). The `Win32_Process` class has some, though; and because WMI method invocation is slightly odd, here's an example:

```
>>> from System import Array
>>> StartupClass = ManagementClass('Win32_Processes')
>>> processes = StartupClass.GetInstances()
>>> proc = list(processes)[-1]
>>> proc.Properties['Name'].Value
'csrss.exe'
>>> arg_array = Array.CreateInstance(object, 2)
>>> proc.InvokeMethod('GetOwner', arg_array)
0
>>> arg_array
System.String[]('SYSTEM', 'NT AUTHORITY')
```

You can see from the `GetOwner` method documentation[9] that it takes two strings as arguments. (The documentation also specifies the meaning of the return value—in this case, 0 for success.) These are `out` parameters to be populated with the user who owns the process and the domain under which it's running. *But*, because the arguments have to be supplied as an array, you can create a fresh array with two members and pass it into `InvokeMethod` along with the method name.

Another method on `Win32_Process` is `SetPriority`.[10] This takes a single integer (the priority) as an argument (64 for idle priority), which you put in an object array.

```
>>> arg_array = Array[object]((64,))
>>> proc.InvokeMethod('SetPriority', arg_array)
0
```

You'll see shortly that PowerShell can make it easier to discover the methods on WMI objects.

A lot of the real power of WMI for system administrators is in the ability to connect to computers on the network. Because this isn't something we've covered yet, let's see how it's done.

10.2.2 Connecting to remote computers

Here's where it starts to get fun. Connecting remotely isn't something you want to allow any old soul to do, and so the security permissions have to be set correctly on the target computer. There are a couple of places where you might have to adjust permissions. To allow remote access, the first place to try is Console Root > Component Services > My Computer > (right-click) Properties > COM Security from the DCOMCNFG application.[11] You can launch DCOMCNFG from the command line, and it should look like figure 10.2.

---

9 See http://msdn2.microsoft.com/en-us/library/aa390460(VS.85).aspx.

10 See http://msdn2.microsoft.com/en-us/library/aa393587(VS.85).aspx.

11 See this page for the details: http://msdn2.microsoft.com/en-us/library/aa393266(VS.85).aspx.

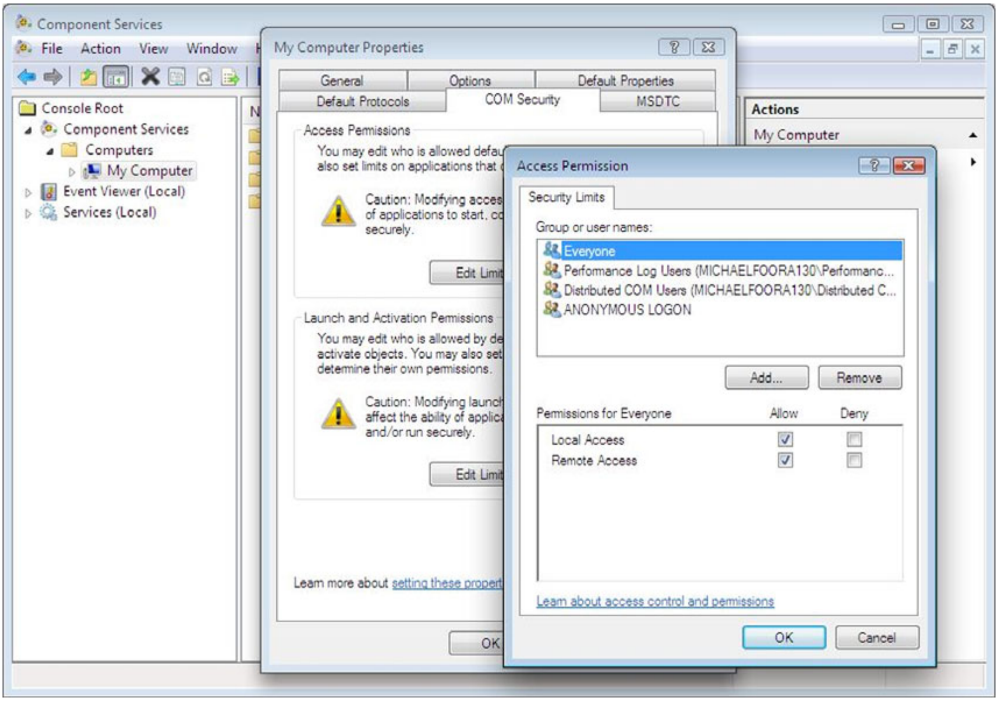


Figure 10.2 Configuring remote access from Component Services

If you still get access permission errors in any of the following examples, you can also set the access permissions for individual WMI namespaces via the Computer Management console from the Control Panel. The full route to this dialog is Control Panel > Administrative Tools > Computer Management > Services & Applications > WMI Control > (right-click) Properties, and it should look like figure 10.3!

We haven't talked about WMI namespaces at all yet. All the examples we've looked at so far have worked without specifying an explicit scope. This means that they've connected to the default namespace on the local machine. To connect to machines on a network, you'll need to connect to an explicit scope.

The default scope is `\\localhost\root\cimv2`, which means the `root\cimv2` namespace on the local machine. CIMV2 (where CIM stands for Common Information Model) is the default namespace and contains all the most commonly used classes, including all the ones we've used so far. There are other namespaces such as `root\DEFAULT`, which contains classes for working with the registry. Other providers can register namespaces to provide instrumentation via WMI. The BizTalk namespace is `root\MicrosoftBizTalkServer`, SQLServer is `root\Microsoft\SqlServer\`, and so on.[12]

[12] There's a recipe in the IronPython Cookbook that will list all the available WMI namespaces and the classes they contain. See http://www.ironpython.info/index.php/WMI_with_IronPython.

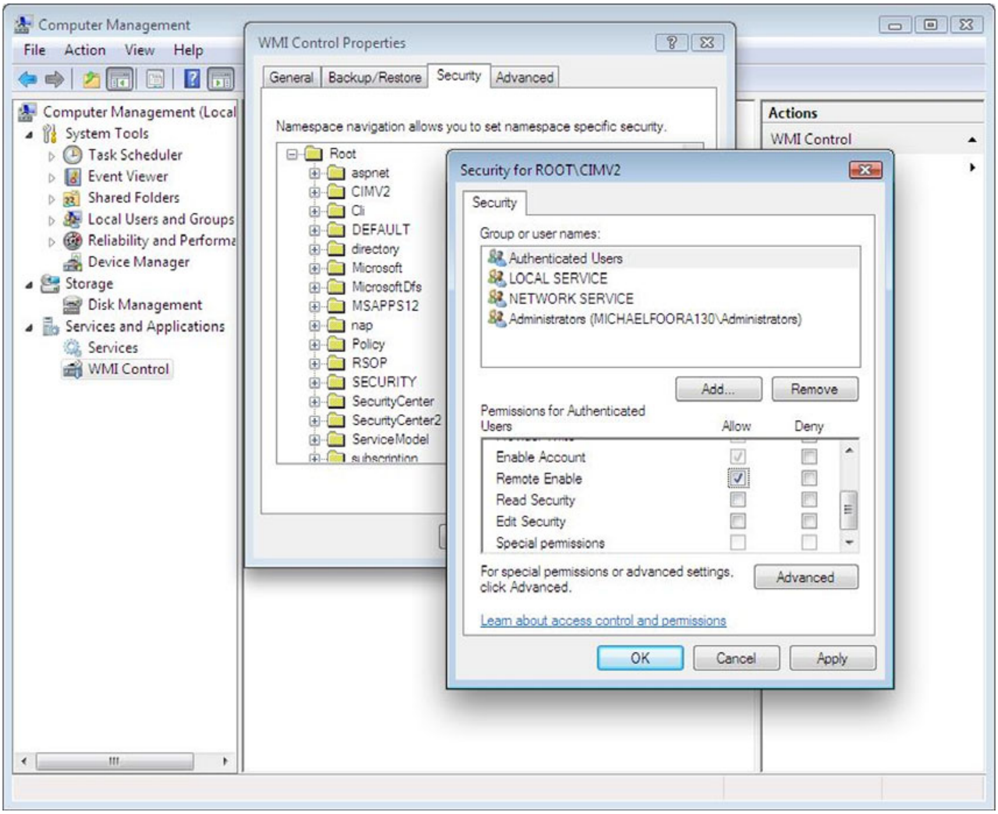


Figure 10.3 Configuring WMI access through Computer Management

To specify the default namespace on a remote machine (in the same domain on the network), you specify a scope like `\\FullComputerName\root\cimv2`. You do this with the .NET `ManagementScope` class.

CONNECTION AUTHENTICATION AND IMPERSONATION

We've already talked about how you enable permissions for remote connections, but you have two choices about how to connect. You can either connect using the credentials of the user running the script, called *impersonation*, or you can explicitly specify a username and password for the connection.

Listing 10.10 shows how to create a `ManagementScope` for a connection to a remote computer with a specific username and password.

Listing 10.10 Specifying username and password for a WMI connection

```
from System.Management import (
   ConnectionOptions, ManagementScope
)
options = ConnectionOptions()
options.EnablePrivileges = True
```

```
options.Username = "administrator"
options.Password = "******"
network_scope = r"\\FullComputerName\root\cimv2"
scope =  ManagementScope(network_scope, options)
```

Listing 10.11 shows how to make the same connection using impersonation.

Listing 10.11 A WMI connection with impersonation

```
from System.Management import (
   AuthenticationLevel, ImpersonationLevel,
   ManagementScope, ConnectionOptions
)
options = ConnectionOptions()
options.EnablePrivileges = True
options.Impersonation = ImpersonationLevel.Impersonate
options.Authentication = AuthenticationLevel.Default
network_scope = r"\\FullComputerName\root\cimv2"
scope =  ManagementScope(network_scope, options)
```

Whether you should use authentication or impersonation depends on the details of the network you're working with. If the computers you're connecting to are configured to allow remote connections from any user with the correct privileges, then impersonation is easier. If the computer limits connections to a specific user, or set of users, then you'll need to use authentication.

QUERYING REMOTE COMPUTERS

Having created the scope, you use it to create a `ManagementEventWatcher` and start listening for events. Listing 10.12 is more of a real-world example than some of the examples we've used so far. It monitors a remote computer for low memory situations (specifically when the available physical memory drops below 10 MB).

Listing 10.12 Monitoring memory use on a remote computer

```
import clr
clr.AddReference('System.Management')
from System.Management import (
   ConnectionOptions, ManagementScope,
   WqlEventQuery, ManagementEventWatcher
)
from System import TimeSpan
from System.Threading import Thread

options = ConnectionOptions()
options.Username = "administrator"
options.Password = "******"
network_scope = r"\\FullComputerName\root\cimv2"
scope =  ManagementScope(network_scope, options)

wql = ('TargetInstance ISA "Win32_OperatingSystem" AND '
     'TargetInstance.FreePhysicalMemory < 10000')

timeout = TimeSpan(0, 0, 5)
query = WqlEventQuery("__InstanceModificationEvent", timeout, wql)

watcher = ManagementEventWatcher()
```

```
watcher.Query = query
watcher.Scope = scope     <—— Specifies scope for query

interesting_properties = (
   'FreePhysicalMemory',
   'FreeSpaceInPagingFiles',
   'FreeVirtualMemory',
   'NumberOfProcesses',
   'SizeStoredInPagingFiles',
   'TotalVirtualMemorySize',
   'TotalVisibleMemorySize',
   'LocalDateTime'
)

def arrived(sender, event):
   print 'Event arrived'
   real_event = event.NewEvent
   instance = real_event['TargetInstance']

   for prop in interesting_properties:
      entry = instance.Properties[prop]
      print entry.Name, entry.Value

watcher.EventArrived += arrived
watcher.Start()

print 'started'
while True:
   Thread.CurrentThread.Join(1000)
```

If you're monitoring a network of servers, you're going to be interested in (and concerned about) events like this. Because you're monitoring for a change in the system, this event is an `__InstanceModificationEvent`, and the WQL is as follows:

```
TargetInstance ISA "Win32_OperatingSystem"
AND TargetInstance.FreePhysicalMemory < 10000
```

Another useful thing to watch for[13] might be disk space dropping below a certain threshold on any fixed disk (that is, not including USB sticks/CDs and so on). Here's WQL with the threshold set at 1 MB:

```
TargetInstance ISA 'Win32_LogicalDisk' AND TargetInstance.DriveType = 3
AND TargetInstance.FreeSpace < 1000000
```

(You could achieve a similar goal by watching for the extrinsic event `Win32_VolumeChangeEvent`.)

To be notified if CPU usage goes above 80 percent on any processor, the WQL is as follows:

```
TargetInstance ISA 'Win32_Processor' AND TargetInstance.LoadPercentage > 80
```

The next query monitors for unauthorized access (failed login attempts). This query relies on access auditing being in place so that the entries go into the event logs. To

[13] *Many* thanks to Tim Golden, a Python and WMI guru, for his help with these examples. Tim has created an excellent module for using WMI from CPython. See http://timgolden.me.uk/python/wmi.html.

remotely access the security logs, you'll need to specify the security privilege. Setting `options.EnablePrivileges = True` should be enough; but, if you're using authentication, then you may need to set `options.Authentication = AuthenticationLevel.Security`. This event is an `__InstanceCreationEvent`, and the WQL is as follows:

```
TargetInstance ISA 'Win32_NTLogEvent' AND
TargetInstance.CategoryString = 'Logon/Logoff' AND TargetInstance.Type =
'audit failure'
```

Systems administration requires a great many tools for different situations. Despite its baroque interface, WMI is an extremely powerful tool. Because of the level of integration with .NET through the managed APIs, WMI works very well with IronPython. In exploring those APIs, we've uncovered quite a few different ways it can be useful, whether you're investigating a single machine or monitoring a whole network of computers. The advantage of Python here is that, as well as rapidly creating simple diagnostic scripts (or even working interactively), you can also build larger monitoring applications where WMI is only a small part of the whole solution.

Another useful tool for Windows system administration is PowerShell. It's more commonly used as a standalone environment, but we're going to look at how IronPython can be part of the answer from inside PowerShell and how PowerShell can become another component for use in IronPython.

10.3 PowerShell and IronPython

The (relatively) new kid on the block for system administrators is PowerShell. PowerShell extends the concept of shell scripting to allow you to pipe objects between commands instead of just data. It's essentially a programming language (cleverly disguised as a scripting environment) specialized for Windows system administration. We know what you're thinking; you have Python—why would you need another language?

NOTE There's an open source implementation of PowerShell for Mono called Pash (PowerShell + bash). See http://pash.sourceforge.net/ for more details. It aims to be a faithful implementation of PowerShell, with the project page proclaiming *the user experience should be seamless for people who are used to Windows' version of PowerShell. The scripts, cmdlets and providers should runs AS-IS* (except where they use Windows-specific functionality).

In this section, you'll see that IronPython and PowerShell can interact in two different ways. We use PowerShell commands and APIs directly from IronPython, and we also use IronPython in PowerShell as a way of overcoming some of PowerShell's limitations.

10.3.1 Using PowerShell from IronPython

The normal way to use PowerShell is as a replacement command line. Running PowerShell opens a console window that looks much like the normal Windows command prompt, `cmd.exe`, but is in fact much more like the Python interactive interpreter. You execute PowerShell commands that return objects, which you can store or pipe to other commands. You can see the PowerShell command prompt in figure 10.4.

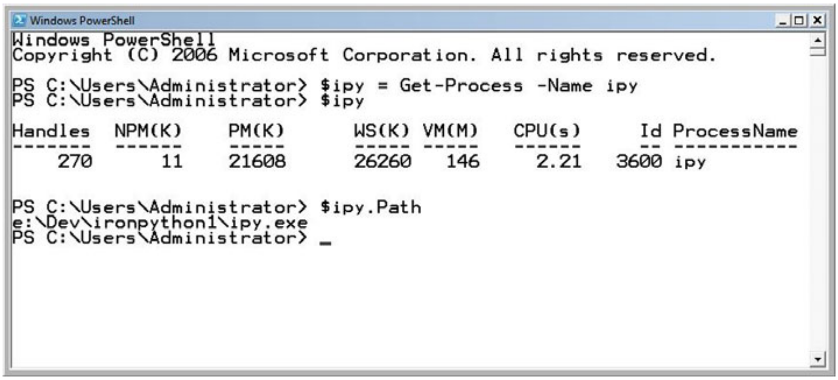


Figure 10.4 The PowerShell interactive environment

PowerShell processes the output of its commands as .NET objects. The commands themselves (cmdlets) are usually thin wrappers around .NET classes. The PowerShell infrastructure provides argument parsing and binding, a runtime, and utilities for formatting and displaying results. This infrastructure is provided through a set of .NET assemblies installed when you install PowerShell. The top-level namespace for this infrastructure and its accompanying APIs is `System.Management.Automation`.[14] This is an apposite name. Automation is at the heart of systems administration. Humans are unreliable and the more we can automate, and keep humans out of the process, the better. Naturally, these namespace are available to use from IronPython.

NOTE To follow these examples, you'll need PowerShell 1.0 installed.[15] This section isn't a comprehensive introduction to PowerShell. If you want to learn more about PowerShell, then *Windows PowerShell in Action* by Bruce Payette (Manning, 2007) is a great resource.

The simplest way to access PowerShell functionality from IronPython is by creating a runspace, which is a kind of execution scope for PowerShell commands. The PowerShell commands live in a different namespace, `Microsoft.Powershell.Commands`. You can use a runspace to execute commands by name, and *don't* need to directly reference this namespace.

THE POWERSHELL RUNSPACE

Listing 10.13 invokes a PowerShell command in a runspace and uses the object that the command returns.

Listing 10.13 Executing PowerShell commands from IronPython

```
import clr
clr.AddReference('System.Management.Automation')
from System.Management.Automation import RunspaceInvoke
```

[14] See http://msdn2.microsoft.com/en-us/library/system.management.automation.aspx.
[15] PowerShell can be obtained from http://www.microsoft.com/powershell.

```
runspace = RunspaceInvoke()
psobjects = runspace.Invoke("Get-Process -Name ipy")     <- Executes command
process = psobjects[0]                                   <- Pulls out first result

print 'Path = ', process.Properties['Path'].Value

for prop in process.Properties:
   name = prop.Name
   if name in ('ExitCode', 'ExitTime', 'StandardIn',
          'StandardOut', 'StandardInput',
          'StandardOutput', 'StandardError'):
      # Can't fetch these on a process
      # that hasn't exited or redirected
      # the in/out/error streams
      continue                     <- Skips properties that raise errors
   print prop.Name, prop.Value
```

IronPython, PowerShell, and COM

Automation with IronPython and COM is a big topic that we don't have the space to cover.[16] As well as using PowerShell for easy access to WMI, you can use it to work with COM. The following snippet shows how to use COM from PowerShell to sync an iPod with the iTunes application:

```
PS > $app = Get-Object -ComObject iTunes.application
PS > $app.UpdateIPod()
```

The call to `Invoke` returns a collection of `PSObject` objects, which you can interact with. One use case is to take advantage of the WMI/PowerShell integration, which can make it easier to work with certain aspects of WMI. Listing 10.14 uses the `Get-WmiObject` command to examine the video controller and the CPU and to find a running process.

Listing 10.14 WMI from PowerShell inside IronPython!

```
import clr
clr.AddReference('System.Management.Automation')
from System.Management.Automation import (
   PSMethod, RunspaceInvoke
)

runspace = RunspaceInvoke()
cmd = "Get-WmiObject Win32_VideoController"
psobjects = runspace.Invoke(cmd)
video = psobjects[0]

print
print 'Video controller properties'
for prop in video.Properties:
   print prop.Name, prop.Value
```

[16] There are several good examples of using COM from IronPython on the IronPython Cookbook, including a good introduction, at http://www.ironpython.info/index.php/Interop_introduction.

```
psobjects = runspace.Invoke("Get-WmiObject Win32_Processor")
cpu = psobjects[0]

print
print 'CPU properties'
for prop in cpu.Properties:
   print prop.Name, prop.Value

cmd = 'Get-WmiObject Win32_Process -filter \'Name="ipy.exe"\''   <- A WMI query from PowerShell
psobjects = runspace.Invoke(cmd)
ipy = psobjects[0]

print
print 'WMI process methods'
for member in ipy.Members:                   <- Loop through all members
   if not isinstance(member, PSMethod):      <- Find the methods
      continue
   print member
```

You'll notice that the last command uses the `filter` keyword. This is a WMI query that uses PowerShell rather than WQL syntax. Like the WMI objects we've already worked with, PowerShell objects have a `Properties` collection that you can iterate over. They also have `Methods` and `Members` collections. Unfortunately, I got null reference exceptions when accessing the `Methods` collection; *but* you can find methods by iterating over all members and checking for instances of the `PSMethod` type.

The IronPython PowerShell sample

The IronPython team has provided a wrapper around a lot of this functionality in the PowerShell sample.[17] You can directly invoke PowerShell commands on the `shell` object they provide, by calling methods with lowercase command names and underscores instead of dashes.

```
>>> from powershell import shell
>>> shell.get_process('notepad').stop_process()
```

MULTIPLE COMMANDS AND THE PIPELINE

The `RunspaceInvoke` instances are great for executing individual commands, but you can achieve more by creating a pipeline. This gets you, in effect, a PowerShell environment embedded into IronPython. Listing 10.15 creates a pipeline, adds commands to it, and then invokes the whole pipeline.

Listing 10.15 The PowerShell pipeline

```
import clr
clr.AddReference('System.Management.Automation')
from System.Management.Automation.Runspaces import (
   RunspaceFactory
)
```

[17] You can download the samples from the IronPython 2.0 release page on CodePlex.

```
runspace = RunspaceFactory.CreateRunspace()
runspace.Open()

runspace.SessionStateProxy.SetVariable("processName", 'ipy')
pipeline = runspace.CreatePipeline()
pipeline.Commands.AddScript('Get-Process -Name $processName')
pipeline.Commands.Add('Out-String')

results = pipeline.Invoke()
for result in results:
   print result
```

This code uses a different technique to create the runspace—from a factory that returns a `Runspace`[18] instance, which you must `Open` before using it. The runspace also has an `OpenAsync` method, which opens it in another thread.

The code also sets the `processName` variable in the execution environment via the `SessionStateProxy`. These APIs are analogous to the IronPython hosting API, and could be useful if you want to expose a PowerShell scripting environment to your users!

The last command added to the pipeline command collection is the `Out-String` command. This formats the results using the PowerShell pretty printer so that, when you print the results, you get nicely formatted output like the one in figure 10.5.

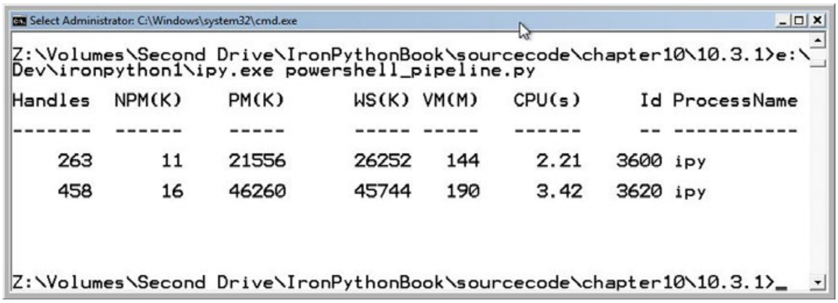


Figure 10.5 The formatted output from a PowerShell pipeline

We've looked at one side of the coin: embedding PowerShell in IronPython. Let's move into the flip side.

10.3.2 Using IronPython from PowerShell

Because PowerShell is a .NET scripting environment, it can use .NET assemblies and objects. The IronPython interpreter is an ordinary (for some value of ordinary) .NET object and can easily be used from other .NET applications, which includes PowerShell.

So why on earth would you want to do this? Well, it turns out that you can use IronPython to overcome certain limitations with PowerShell. These limitations include operations that would block the console or actions that should only be done from an STA thread and don't work directly from PowerShell, which runs in a Multi-Threaded

[18] See http://msdn2.microsoft.com/en-us/library/system.management.automation.runspaces.runspace.aspx.

Apartment (MTA).[19] You can also use IronPython from within PowerShell to work with Python libraries.

EMBEDDING IRONPYTHON IN POWERSHELL

You embed IronPython via its hosting API—which is something we'll explore in more detail when we look at providing a scripting API to a .NET application with IronPython. IronPython 1 and 2 have different hosting APIs, so how you access IronPython from inside PowerShell depends on which version of IronPython you have.

> **Executing the examples**
>
> To execute the example scripts,[20] you'll need to set the execution policy to allow unsigned scripts. The PowerShell command to do this is as follows:
>
> ```
> Set-ExecutionPolicy Unrestricted
> ```
>
> For more information about script signing, you can execute the following command:
>
> ```
> Get-Help About_Signing
> ```

Listing 10.16 shows the PowerShell code necessary for executing code with IronPython 1. It assumes you have the IronPython assemblies in the current working directory.

Listing 10.16 IronPython 1 in PowerShell

```
$full_path = Resolve-Path $cur_dir 'IronPython.dll'
[reflection.assembly]::LoadFrom($full_path)        <— Loads IronPython assembly

$engine = New-Object  IronPython.Hosting.PythonEngine
$engine.Execute("print 'Hello World! from IP1'")     <— Executes code from string
```

The call to load assemblies requires an absolute path, which you construct with a call to `Resolve-Path` (which resolves paths relative to current working directory). Having constructed an IronPython engine, Python code is executed with the `Execute` method.

> **The IronPython 2 hosting API**
>
> The code here is written against the hosting API of IronPython 2.0.
>
> These examples only use a small part of the IronPython hosting API. Chapter 15 has a much more in-depth look at embedding IronPython in other .NET environments, and many of the techniques shown there could also be used from PowerShell.

---

[19] PowerShell 2 will support an `-sta` command-line switch. Even then this solution could be useful because it will allow you to access STA functionality without having to start PowerShell with particular command-line arguments.

[20] Downloaded from http://www.ironpythoninaction.com/, of course.

Listing 10.17 shows the equivalent for code for IronPython 2. The code is more complicated because IronPython 2 is built on the DLR and the hosting API is more generic.

Listing 10.17 IronPython 2 in PowerShell

```
$base_dir_env = Get-Item env:IP2ASSEMBLIES
$base_dir = $base_dir_env.Value
$first_path = Join-Path $base_dir 'Microsoft.Scripting.dll'
$second_path = Join-Path $base_dir 'IronPython.dll'
[reflection.assembly]::LoadFrom($first_path)
[reflection.assembly]::LoadFrom($second_path)

$engine = [ironpython.hosting.python]::CreateEngine()
$st = [microsoft.scripting.sourcecodekind]::Statements
$code = 'print "Hello World from IP2!"'
$source = $engine.CreateScriptSourceFromString($code, $st)
$scope = $engine.CreateScope()
$source.Execute($scope)
```

Fetches IP2ASSEMBLIES environment variable

Turns Python code into ScriptSource

This snippet uses a different technique to load the assemblies. It assumes you've set an environment variable `IP2ASSEMBLIES` with the path to a directory containing the IronPython 2 assemblies.

To execute code you have to create a script source from the code string and the `SourceCodeKind.Statements` enumeration member. The syntax to do this in PowerShell is somewhat ugly. The obvious thing to do is to abstract this little dance out into a function like listing 10.18.

Listing 10.18 Executing Python code from a function in PowerShell

```
$base_dir_env = Get-Item env:IP2ASSEMBLIES
$base_dir = $base_dir_env.Value
$first_path = Join-Path $base_dir 'Microsoft.Scripting.dll'
$second_path = Join-Path $base_dir 'IronPython.dll'

[reflection.assembly]::LoadFrom($first_path)
[reflection.assembly]::LoadFrom($second_path)

$global:engine = [ironpython.hosting.python]::CreateEngine()
$global:st = [microsoft.scripting.sourcecodekind]::Statements

Function global:Execute-Python ($code) {
   $source = $engine.CreateScriptSourceFromString($code, $st)
   $scope = $engine.CreateScope()
   $source.Execute($scope)
}
```

This listing creates a function, which executes code that you pass in as a string. PowerShell's scoping rules are very different from Python's.[21] The `global` keyword makes `Execute-Python` available to the interactive environment when this code is executed from a script. Because PowerShell is dynamically scoped, all the variables the function uses also have to be global because they'll be looked up in the scope that *calls* the function.

[21] And not at all better in our opinion. Dynamic scoping is designed with interactive use in mind, and is the same as the scoping rules used by Bash.

`Execute-Python` is called, as follows:

```
Execute-Python 'print "Hello world from PowerShell"'
```

You can build on this general technique, whether working with IronPython 1 or 2, to do various things useful from within the PowerShell environment.

CREATING STA THREADS

PowerShell runs in an MTA thread, which causes problems for code that has to be called from an STA. This prevents you using Windows Forms objects, such as calling `Clipboard.SetText` to put text on the clipboard. You can get around this by spinning up an STA thread from IronPython and setting the clipboard from there[22] (listing 10.19).

WARNING Unhandled exceptions inside threads will cause PowerShell to bomb out and die! You *will* get the exception traceback when it happens. Running PowerShell from cmd.exe rather than launching it from the start menu will give you a chance to read the traceback.

Listing 10.19 Setting the clipboard from PowerShell with IronPython 1

```
$global:ClipCode = @'
import clr
clr.AddReference("System.Windows.Forms")
from System.Windows.Forms import Clipboard
from System.Threading import (
   ApartmentState, Thread,
   ThreadStart
)

def thread_proc():
    Clipboard.SetText(text)

t = Thread(ThreadStart(thread_proc))
t.ApartmentState = ApartmentState.STA
t.Start()
'@

Function global:Set-Clipboard ($Text){
  $engine.Globals["text"] = $Text
  $engine.Execute($ClipCode)
}
```

This code works with IronPython 1 and assumes you've already created the IronPython engine as the `$engine` variable (and made it global). The reason this code is specific to IronPython 1 is that it sets the `text` variable in the Python engine `Globals` so that the IronPython code can use it to set the text on the clipboard. To make this code work with IronPython 2, you need to create an explicit execution scope and set the variable in there. You then need to pass the scope in when you call `Execute` on `$ClipCode`, and this is where the fun starts.

When you call `Execute` with one argument (a `ScriptScope`), it becomes a generic method. Calling generic methods from PowerShell is non-trivial. Luckily, Lee

[22] Many thanks to Marc, The PowerShell Guy, who provided the original code for this example.

Holmes has solved this problem, so you'll use his `Invoke-GenericMethod`[23] script to invoke `Execute`.

Again assuming that you've already created an IronPython engine, listing 10.20 creates a `Set-Clipboard` function that sets text on the clipboard using IronPython 2.

Listing 10.20 Setting clipboard from PowerShell with IronPython 2

```
$global:scope = $engine.CreateScope()
$global:ClipCode = $engine.CreateScriptSourceFromString(@'
import clr
clr.AddReference("System")
clr.AddReference("mscorlib")
clr.AddReference("System.Windows.Forms")
from System.Windows.Forms import Clipboard
import System
from System.Threading import Thread, ThreadStart

def thread_proc():
    Clipboard.SetText(text)

t = Thread(ThreadStart(thread_proc))
t.ApartmentState = System.Threading.ApartmentState.STA
t.Start()
'@, $st)

Function global:Set-Clipboard ($Text){
  $scope.SetVariable('text', $Text)                    <- Sets text in scope
  $params = @('microsoft.scripting.hosting.scriptscope')       <- Type of parameter for Execute
  ./Invoke-GenericMethod $ClipCode 'Execute' $params $scope    <- Invokes Execute on $ClipCode with $scope
}
```

Another difference between this code and the code for IronPython 1 is that, for IronPython 2, you need to explicitly add references to the system assemblies, both System.dll and mscorlib.dll. In IronPython 1, the `PythonEngine` does this, but not in IronPython 2.

The code that finds the right generic overload of `Execute` isn't pretty, but it's abstracted away in the `Invoke-GenericMethod` script. The call parameters are as follows:

```
./Invoke-GenericMethod instance MethodName params arguments
```

`params` should be an array of strings with the type names of the arguments. The `arguments` parameter is the set of arguments that `Execute` is to be called with, passed in as an array of objects. If you pass in an individual string and an individual object for params and arguments, then PowerShell will cast them into arrays.

ASYNCHRONOUS EVENTS WITHOUT BLOCKING

The next use case for IronPython from PowerShell is for handling events. In .NET, asynchronous events are raised on another thread, preventing you from using PowerShell script blocks as event handlers. The usual solution is to wait for the event to be raised on the main execution thread, which blocks the console. You can get around this by subscribing to the event from IronPython.

---

[23] See http://www.leeholmes.com/blog/InvokingGenericMethodsOnNonGenericClassesInPowerShell.aspx.

Listing 10.21 uses the `EventLog` class,[24] and its `EntryWritten` event, to print the details of any messages written to the Windows event logs.

Listing 10.21 Handling asynchronous events from PowerShell with IronPython

```
$source = $engine.CreateScriptSourceFromString(@'
import clr
clr.AddReference('System')
from System.Diagnostics import EventLog

def handler(sender, event):
   print 'Entry from', sender.Log
   entry = event.Entry
   print entry.Message

logs = EventLog.GetEventLogs()
for log in logs:
   try:
      log.EnableRaisingEvents = True
      log.EntryWritten += handler
      print 'Added handler to', log.Log
   except:
      print 'Failed to add handler to', log.Log
'@, $st)

$scope = $engine.CreateScope()
$source.Execute($scope)
```

After running this code, control returns immediately to the console. To see your event handlers in action, start a new program, or perform any action that causes writes to event logs, and you'll see the log messages appear at the console. You can see the start of one of these messages in figure 10.6.

So far we've been using IronPython to access .NET features from PowerShell. Because PowerShell has native access to most of .NET, bar a few limitations, a more compelling reason to use IronPython is to access Python itself. In particular, you can use IronPython to take advantage of Python libraries.

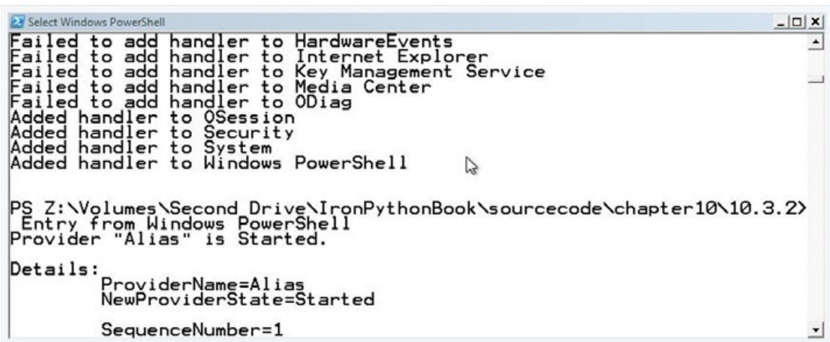


Figure 10.6 Listening to the Windows event logs

[24] See http://msdn2.microsoft.com/en-us/library/system.diagnostics.eventlog.aspx.

CALLING PYTHON CODE AND RETURNING RESULTS

Using the same pattern as the previous examples, you can create a PowerShell function that calls into Python code and returns the result. In theory, you could do this with a single expression, creating the `ScriptSource` with `SourceCodeKind.Expression` rather than `SourceCodeKind.Statements`.[25] Calling generic methods that return values becomes an even bigger world of pain, but there's a simple way around this: you can assign the return value to a variable and fetch that back out of the scope.

The basic pattern is as follows:

```
$src = 'result = some_function(value)'
$script = $engine.CreateScriptSourceFromString($src, $st)
$scope.SetVariable('value', $value)

./Invoke-GenericMethod ...

[Ref] $result = $null
$scope.TryGetVariable('result', $result)
$result.Value
```

Fetching the result out of the scope is done with `TryGetVariable`, which takes an `out` parameter. You do this from PowerShell by creating a `[Ref]` type. You fetch the resulting value by accessing the `Value` property after the call to `TryGetVariable`.

Listing 10.22 pulls all this together. It provides two functions, `B64Encode` and `B64Decode`, that can encode and decode strings with the base64 encoding, using the `base64`[26] library from the Python standard library.

Listing 10.22 Calling Python functions and returning values

```
$setupSrc = @'
import sys
sys.path.append(r'c:\Python25\lib')
import base64
'@                <-- Setup code that imports base64
$init_code = $engine.CreateScriptSourceFromString($setupSrc, $st)

$src = 'result = base64.b64encode(value)'
$global:encode = $engine.CreateScriptSourceFromString($src, $st)
$src = 'result = base64.b64decode(value)'
$global:decode = $engine.CreateScriptSourceFromString($src, $st)

./Invoke-GenericMethod $init_code 'Execute' $params $scope     <-- Executes setup code

Function global:B64Encode ($value){
  $scope.SetVariable('value', $value)
  ./Invoke-GenericMethod $encode 'Execute' $params $scope | out-null
  [Ref] $result = $null
  $scope.TryGetVariable('result', $result) | out-null     <-- Fetches result
  $x.Value          <-- Returns result
}

Function global:B64Decode ($value){
  $scope.SetVariable('value', $value)
```

[25] Assumin that you're working with the IronPython 2 API.

[26] See http://docs.python.org/lib/module-base64.html.

```
  ./Invoke-GenericMethod $decode 'Execute' $params $scope | out-null
  [Ref] $result = $null
  $scope.TryGetVariable('result', $result) | out-null
  $result.Value
}
```

PowerShell functions return all unhandled output. Inside `B64Encode` and `B64Decode`, unneeded values are suppressed by piping them to `out-null`. The real result is returned by `$result.Value`, and it does in fact work!

```
PS C:\> $a = B64Encode 'This really works!'
PS C:\> $a
VGhpcyByZWFsbHkgd29ya3M=
PS C:\> B64Decode $a
This really works!
```

Extending this example to call Python functions that take or return multiple values would be simple—just set and fetch more variables in the scope.

One interesting, if slightly insane, way of using this would be when embedding PowerShell into IronPython. You could pass in a scope populated with Python callback functions, and call into them from PowerShell as a way of communicating between the environments.

PowerShell is an interesting new programming environment. We're not about to give up IronPython for PowerShell, but it's great to see that these two systems can work well together. After summarizing this chapter, we'll move on to using IronPython with a completely different system.

10.4 Summary

Python is a powerful general-purpose programming language, and its combination of clarity and succinctness means that systems administration is an area where it shines. The integration of the .NET framework to the Windows operating system makes IronPython particularly suited to Windows system administration.

Python eats simple scripting tasks for breakfast, but it has the great advantage of scaling well when simple scripts need to grow and become applications. Whatever task you're tackling, you should check for standard library or third-party modules that could help. After the standard library, we recommend the Python Package Index (PyPI)[27] be your first port of call. Equally importantly, if you create general-purpose libraries to support your Python applications, you should consider creating Python packages with `distutils`[28] or `setuptools`[29] for distribution via PyPI.

For systems administration, both WMI and PowerShell can also be powerful tools. Despite the oddness of WMI, it provides a high-level API for working with low-level details of a system, such as the BIOS, the computer hardware, and the operating

[27] The Python package repository. See http://pypi.python.org/pypi.

[28] The standard library module for compiling, creating, and installing Python packages. See http://docs.python.org/lib/module-distutils.html.

[29] `setuptools` is a third-party framework for the easy installation of Python packages. See http://peak.telecommunity.com/DevCenter/setuptools.

system. For networked operations, it often provides a ready-made solution where the alternative would be a custom-written application.

The topic of PowerShell is a slight anomaly for this book. PowerShell is an alternative programming language; but, for programming tasks of any size, IronPython is more suitable. This is hardly surprising, though; PowerShell is highly specialized to provide a scripting environment for admins rather than to be an application programming language. Despite their different virtues, the two environments can work well together—IronPython using PowerShell for the things it's good at and vice versa.

The next chapter is on a very different topic: web application programming with IronPython and ASP.NET.

11 IronPython and ASP.NET

This chapter covers

- ASP.NET concepts
- Building a web-based document editor
- Handling view state
- Creating reusable controls

So far we've been focusing on using IronPython with Windows Forms to construct rich desktop applications, but the .NET platform also provides a high-level framework for building web applications: ASP.NET.

ASP.NET includes lots of goodies for web development. It provides a powerful page model where server-to-client round-tripping of data is handled automatically, removing the need for a lot of the boilerplate code in a web application. The .NET class library has a huge range of classes for solving different problems, and ASP.NET also comes with a large number of its own built-in components for user interfaces. The end result is that ASP.NET can help you write web applications faster and with less code—and it can do this even better once we add Python to the mix!

The IronPython team has released a project enabling the use of IronPython within ASP.NET. In this chapter we use it to build a simple application to display a MultiDoc file in a web page, and then extend it to allow you to edit the file over the web. Finally you'll see how you can package up things we've built as controls that can be used in other web pages. Before taking a closer look at the IronPython integration, though, let's get an overview of the framework itself.

11.1 Introducing ASP.NET

When writing web applications, the user interface isn't defined in terms of Windows Forms controls, but instead is displayed by the web browser. This means that your interface needs to be represented to the browser as HTML and JavaScript.

> **Clients and servers**
>
> Web applications are networked applications—they're implemented as a conversation between two computers. When we talk about events that occur in the application, it's important to have a clear understanding of where the events happen. Generally, the computer running the web browser that displays the application user interface is called the *client*, and events that happen on that computer are *client-side* events. The computer that lives at the browser's destination URL is called the server, and things that happen on the server are said to be *server-side*.

ASP stands for *Active Server Pages*, and was introduced by Microsoft with Internet Information Services (IIS) 3 in 1996. Originally, it provided a simple way of intermingling HTML and JavaScript client-side code with code (usually written in a language called VBScript) executed on the server. When the .NET platform was released in 2002, it introduced a model for building web applications that was substantially different from the original ASP system, called ASP.NET.

ASP.NET provides a structure for creating web applications that allows you to reuse parts of your user interface in the same way that you can reuse code. It also enables you to package the client-side definition of components (their HTML and JavaScript code) together with code that defines their behavior on the server. Before we can look at how ASP.NET combines the client and server code, we need to define some of the key concepts of the system.

11.1.1 Web controls

Web controls are .NET classes used by the ASP.NET machinery to generate HTML and JavaScript code; they're the building blocks of web pages. The programming interface they expose is designed to be similar to the Windows Forms controls, although there are many differences because of the more constrained request-response model of the World Wide Web. ASP.NET includes dozens of web controls, from the common `Button` and `TextBox` to the powerful `GridView`, and it's easy to write new ones yourself.

An important point to note is that some web controls, such as the `Panel` and `Repeater`, act as containers that can hold other controls. They enable you to construct a user interface as a *tree* of controls, with containers providing structure for the text boxes, labels, check boxes, and buttons.

11.1.2 Pages and user controls

Web controls are put together using a language that is HTML with extensions to allow adding server code and referring to web controls. Files in this language can be pages (with the .ASPX extension), or they can be reusable components called user controls (in ASCX files), which can then be referenced from ASPX pages or other user controls. Both ASPX pages and user controls can have server code either in the same file or in an associated file, called its code file or code-behind. Separating the page and server code is generally cleaner and easier to read, especially if the behavior of the page or control is complicated.

The extensions that make an ASPX page more than HTML include the following:

- *Directives*—Located at the top of a file. They tell the ASP.NET system how to process the file: where the code file is, whether the page should be cached, as well as a host of other options. They can also add references to user controls that can then be used in the page.
- *The `runat="server"` attribute*—Added to HTML elements. When an element has this attribute, an object representing it is created on the server when the page is requested. Server code for the page can interact with it to change its contents or appearance, or even hide it.
- *Web controls*—Included as tags, which look like `<asp:Button id="button1" runat="server" text="Click"/>`. The name `button1` can then be used from the server code to interact with the button. Attribute values (like the button's text) can also be set in the tag.
- *Code snippets*—Included in the page, enclosed by `<% %>` tags. If the snippet starts with `=`, the value of the expression will be converted to a string and appear in the HTML sent to the client.

11.1.3 Rendering, server code, and the page lifecycle

So an ASP.NET page is a tree of web controls, constructed according to ASPX and ASCX files. Each web control in the tree knows how to convert itself into HTML and JavaScript code to be displayed in a browser. Where does the code-behind associated with pages and user controls fit into the picture?

The server code is used in two ways. The simplest way is that code-behind can provide methods that can be called from code snippets in the page or user control. This technique can be useful for displaying small pieces of text or turning a part of the markup on or off.

The second way is much more central to the power of the ASP.NET framework. When a page is requested and the tree of controls is being built, the system fires a number of

different events that your server code can hook into. Some of these events, such as the `Page_Load` event, are fired every time a request for the page is made; others, such as button clicks or drop-down list selection change events, are only raised when triggered by the user on the other end of the internet. These events provide a tremendous amount of control over how the page structure is created and how the controls in the page are set up. They also make it much simpler (from the developer's point of view) to handle user interaction in a web environment. Figure 11.1 shows some of the events fired by the server while handling a request.

Now that you've seen a little of what ASP.NET is and what it has to offer, we add IronPython into the mix and use it to create some ASP.NET pages.

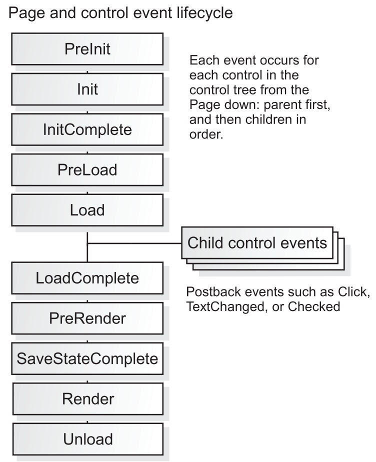


Figure 11.1 The ASP.NET page lifecycle

11.2 Adding IronPython to ASP.NET

Before you can start creating a simple IronPython web application, you'll need to download two things: Visual Web Developer Express (the free Microsoft IDE for ASP.NET) and the IronPython ASP.NET integration package.

Visual Web Developer Express is available from the following URL:

http://msdn.microsoft.com/vstudio/express/downloads/default.aspx.

You should install it with all the default options. Including the MSDN Library documentation can be useful if you want to refer to a local copy, although it makes the download bigger.

IronPython for ASP.NET is available from the following URL:

http://www.codeplex.com/aspnet/Release/ProjectReleases.aspx?ReleaseId=17613.

Download the ASP.NET WebForms IronPython Sample and the documentation package, and unzip them where convenient. The Sample zip file contains the directory layout and files needed for an IronPython ASP.NET web project. We use this as a template when creating our web application.

Why do you need the IronPython for ASP.NET package? You already have IronPython installed, and ASP.NET allows you to use any .NET language, right? Well, not quite. Although ASP.NET is designed to be able to handle many different programming languages, it expects all of them to compile into normal .NET assemblies; as you've seen already, this is something that IronPython doesn't do because its object model is so different from that of the .NET platform.

To solve this problem, the IronPython team has come up with an alternative compilation and execution model for pages written in Python (and one likely to be extended

to other dynamic languages in the future). You can see the differences in figure 11.2.

For more details about the changes made to accommodate dynamic languages, it's well worth reading David Ebbo's whitepaper, available at http://www.asp.net/ironpython/whitepaper/. This covers the changes made to ASP.NET internals and looks at the performance implications of the new model.

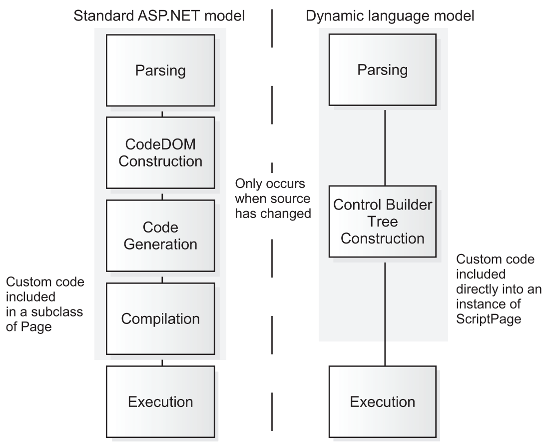


Figure 11.2 Changes to the compilation model in ASP.NET for IronPython

From the perspective of a programmer writing web applications using the IronPython support, the biggest difference lies in the way custom page code is incorporated. In a standard ASP.NET page, your methods and code snippets live in a subclass of the built-in `Page` class. In an IronPython page, you don't define a new class—your code is run by an instance of `ScriptPage` created at execution time. This change has some implications for how you manage the page state in your applications, as you'll see later in the chapter.

11.2.1 Writing a first application

Now that you have all the prerequisites, you're ready to create an IronPython web project by following these steps:

1. Copy the ironpython-webform-sample directory from where you unzipped it, and rename it to ironpython-in-action.
2. Start Visual Web Developer and select the File > Open Web Site… menu option.
3. Navigate to the new ironpython-in-action directory in the dialog box and click Open.

You'll see the directory structure in the Solution Explorer panel on the right side of the window (figure 11.3).

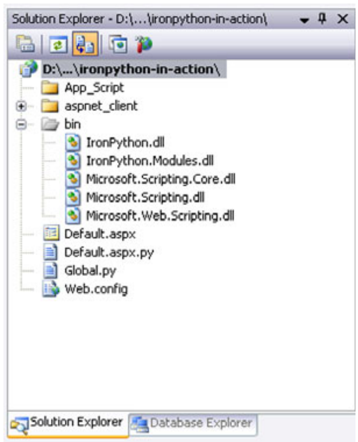


Figure 11.3 The skeleton web application structure in Visual Web Developer

Now that the site has been created, you can edit the files to add some behavior. Open Default.aspx by double-clicking it in the Solution Explorer. This is the home page of the application—the word *default* in the filename means that this page will be shown if someone makes a request for the root directory of your web application. As you can see, the markup in the file looks

much like Extensible Hypertext Markup Language (XHTML), but note the following two important differences:

- *The first line is an ASP.NET directive that tells the framework that any code in this file will be in IronPython and that the code-behind for this page is Default.aspx.py.*

  A code-behind file stores the code for this specific page; keeping the ASPX markup and code separate makes them easier to maintain.
- *The* `<head>` *and* `<form>` *tags have* `runat="server"` *attributes.*

  This attribute tells the ASP.NET machinery to parse the contents of these tags and execute any web controls encountered.

If you click the Design button at the bottom of the window, you can see that the page body contains an `asp:Literal` element. Let's change that—click the literal and delete it, and then drag a `Label` onto the page from the Toolbox tab on the left. If you change back to source view by clicking the Source button, you can see that the IDE has added the following ASPX code to create a `Label` in the body of the page:

```
<asp:Label id="Label1" runat="server" text="Label"></asp:Label>
```

Let's break this down into its separate components:

- *`asp:Label`*—Tells ASP.NET the class of the web control that should be inserted.
- *`id="Label1"`*—Gives the control a name so that you can refer to it from code.
- *`runat="server"`*—Tells ASP.NET that this is a control that it should handle. (Without this attribute, the tag would be ignored and sent directly to the browser as is.)
- *`text="Label"`*—Is a property of the `Label` web control that determines what it will display.

Edit the `asp:Label` tag to set the `Text` property to `""` instead of the default text Visual Studio has created, and save the file.

The project skeleton has several other things in the directory—which we'll look at soon; the most important one for now is the Default.aspx.py file, which is the code-behind file for Default.aspx page. Open this file and you'll see the definition of a function called `Page_Load`. This function is what the processing machinery calls when the page is first requested; it passes in the page as `sender` and any event arguments as `e`. At the moment, it sets the text of the old literal element you removed.

Change the `Page_Load` function to the following:

```
def Page_Load(sender, e):
   Label1.Text = "Hello from IronPython"
```

Make sure that the second line is indented so that it's the body of the function. Save the file.

Now a little magic: click the Start Debugging button (which looks like a green Play button) in the toolbar. By default, the project isn't configured for debugging, so Visual Web Developer will ask whether you want the configuration changed—say Yes. Then it will start its built-in development web server on our new project, and launch a

browser window navigating to the current page. After a brief wait (and maybe a notification from your firewall software asking whether the web server should be allowed to listen for requests), you should see the *Hello from IronPython* message you entered in the `Page_Load` function.

> **Debugging**
>
> The debugger integration for ASP.NET and IronPython is impressive; you can place breakpoints by clicking in the gray gutter to the left of the code lines. Then, when you click the Play button (or hit F5) and the web browser window appears, execution will pause at the breakpoint. You can inspect the values of variables by hovering over them, and step through single lines of code or into function calls. The debugger can make it much easier to work out what your application is doing, particularly when it's not what you expect.
>
> The integration with IronPython isn't totally seamless at the moment, though; the variable values displayed aren't quite right, because they're displayed at the C# level rather than at the IronPython level, but it's still useful.
>
> Here are a few potentially useful tips:
>
> - In a multipage application, you can set the default page shown in the browser when you debug by right-clicking a page in the project and selecting Set As Start Page.
> - Breakpoints on the lines of def statements that define functions aren't part of the function, so they won't be hit when the function is called; you need to put them on lines inside a function to debug calls to it.
> - If you want to view the page you've been editing in the browser without stopping at breakpoints, you can right-click the page in the project and select View In Browser. This will start the integrated web server, but won't switch Web Developer into debugging mode. Ctrl-F5 does the same thing, except uses the start page that has been set for the project.

That's the first step: a basic page generated from IronPython code. Now let's see how to have the page respond to user input.

11.2.2 Handling an event

At this point you have code in the code-behind file that interacts with a control in the ASPX page. Although this technically is handling an event (the `Load` event of the `Page` class), extending the page functionality slightly demonstrates a little more of the power of the ASP.NET system. Let's add some controls to our page: a text box where users can enter their names and a button that will trigger some processing on the server.

Switch back to editing the ASPX page, and add the following lines at the start of the main `<div>`, before the `Label`:

```
Name:
<asp:TextBox id="TextBox1" runat="server"></asp:TextBox>
<asp:Button id="Button1" runat="server" text="Go!" onclick="Button1_Click"
/><br />
```

You can add the controls by typing them into the Source view, or by dragging them from the toolbox in either Source or Design view. Change the text on the button to something more interesting—like Click Me or Go!—something with a bit of vim. Add the `onclick` attribute to the button; this tells ASP.NET to call the function `Button1_Click` when the button is clicked.

Now edit the code-behind page to have the following function definitions:

```
def Page_Load(sender, e):
   pass
def Button1_Click(sender, e):
   Label1.Text = "Hello, " + TextBox1.Text
```

Click the Start Debugging button again; and, after a small pause, you should see a web page with a text box and a button. If you type some text into the text box and click the button, the message *Hello, <name you typed>* will appear below the text box.

So what's happening? Here are a few points to think about:

- There are two requests for the page: the first when the page is initially loaded, and the second when you click the button.
- When the page is submitted because of the button click, the ASP.NET machinery keeps track of this and ensures that the `Button1_Click` function is called.
- After the second request, when the *Hello, <name>* message is visible, the text box is still populated, even though you've done nothing on the server side to set its value. The web controls' state is stored in a location called the *view state*, and automatically restored by the framework before event handlers like `Button1_Click` are called.

Later on in the chapter, we explore how view-state handling works by making something more complex: an application that can display and edit MultiDoc documents. We reuse the MultiDoc modules you already have to create, load, and save documents. To integrate the modules into the project, we need to look at some of the support infrastructure ASP.NET provides.

11.3 ASP.NET infrastructure

You may have noticed several other items in the project directory we used as a template: files called Web.config and Global.py, along with the App_Script , aspnet_client, and bin folders.

The bin folder contains the IronPython assemblies, as well as Microsoft.Web.Scripting.dll, which provides the integration between ASP.NET and IronPython. The aspnet_client directory is where any supporting JavaScript needed by the ASP.NET controls will be placed. The other three items provide ways of hooking into the ASP.NET system, and each one allows you to customize the environment in different ways.

11.3.1 The App_Script folder

For code in our web application to use the classes you've already written, you need to be able to import the modules. In the normal Python world, you can make one module available for import by another one in the following three ways:

- Putting the library module in the same directory as the importing module
- Putting the library module in a directory on the Python import path (sys.path)
- Adding the directory of the library module to the import path

These approaches all work just as well with IronPython in general; but, in the ASP.NET environment, the first option isn't available (because the current directory is generally the location of the web server, rather than the directory containing the currently executing page), and the second isn't useful because, by default, sys.path is empty. Instead, the system provides a special folder called App_Script; any packages and modules stored in this folder will be available for import by any other Python code in the web application. (The third option is still available to you, as you'll see in section 11.3.2.)

Several other special ASP.NET folders can be added to the project, if they're needed, by right-clicking the project in Web Developer and selecting the Add ASP.NET Folder option. The most important of these folders is the App_Code folder, which is similar to the App_Script folder but is intended to store C# (or other compiled language) library code. This code will then be available to all modules in the web project, along with the extra feature that the ASP.NET system will automatically detect changes to the files and recompile them when necessary.

Because we want to be able to read and write MultiDoc files, you need to add the modules for those tasks to the web project. To do this, right-click the App_Script folder and select Add Existing Item to add the files model.py, xmldocumentreader.py, documentreader.py, and documentwriter.py from chapter 5 to the project. Once this is done, you'll be able to use them to read and write MultiDocs from any code-behind page in the web application.

11.3.2 The Global.py file

The App_Script folder gives you a place to keep support modules that are used by pages in our project, but it's really meant for library code that you'll write specifically for *this* project; if several projects required the same module, you'd need to copy that Python file to each App_Script folder. A better solution in this situation is to put the module in a different location outside your project, and add that project to the `sys.path` list.

You could modify `sys.path` by adding a line in your code-behind files, but you'd need to do so on every ASPX page because you don't necessarily know which page will be visited first. The Global.py file can help because it provides a way to listen for events that affect the application, no matter which page requested to cause the event.

Looking at the Global.py file, you can see that it contains the following empty event handlers:

- *`Application_Start`*—Occurs when your application receives its first request and is loaded to respond.
- *`Application_End`*—Happens when the last session created by a request to your application ends, and the ASP.NET framework unloads it. Subsequent requests to the application will trigger another `Application_Start` event.

- *Application_Error*—Happens when an uncaught exception is thrown during a page request, allowing a standard mechanism for logging or changing error display.
- *Application_BeginRequest*—Is fired before each request is processed by your application.
- *Application_EndRequest*—Is fired after each request processed by your application.

In our case, if we want to have extra directories in `sys.path`, the `Application_Start` event handler is a good place to add them. Because the MultiDoc modules rely on modules in the Python standard library, you should add the `Lib` directory to the `sys.path` with the following change:

```
def Application_Start():
   import sys
   sys.path.append("C:\\Python25\\Lib")
```

(This code assumes that Python is installed in C:\Python25; change the path accordingly if not.)

11.3.3 The Web.config file

The .NET framework provides a comprehensive, extensible configuration system based on an XML format and a wide range of settings for web projects. The skeleton project we've used contains a Web.config file for the project with the configuration for our application. In particular, it includes a number of directives that enable IronPython support in the application. For the moment, you don't need to change the Web.config file—the default settings are fine for what we're going to do next.

That's all the setting-up finished; now we can implement the MultiDoc Viewer using the modules we've added to the project.

11.4 A web-based MultiDoc Viewer

We start by creating a page that can display the contents of a MultiDoc, and then in section 11.5 we extend it with controls and code to allow you to edit the file. To begin with, the page will display a sample static MultiDoc XML file stored in the web application directory; later we explore how you can change the Viewer to accept the document to display as a parameter.

For the sample MultiDoc file, add a new text file called doc.xml to the project, and populate it with the following XML:

```
<document>
   <page title="Page one">This page is the first in the MultiDoc</page>
   <page title="Page two">Welcome to the heady delights of the second page</
     page>
</document>
```

In the following sections we make the ASPX code and Python code-behind that will display the information in this XML.

11.4.1 Page structure

First, we need to decide how to display a MultiDoc object. Each document is essentially a list of pages—we could display each page in order, but that would be quite different from the tab pages the desktop application uses for display. (It also wouldn't illustrate some of the details that we want to explore in the ASP.NET system.)

Instead, let's display a MultiDoc as a list of the titles of all the pages it contains, rendered as links that can be clicked on to display the contents of that page. Then the contents of the selected page will be displayed as a title and body (figure 11.4).



Figure 11.4 Sketch of the page layout for the MultiDoc Viewer

We can convert this sketch into ASPX code piece by piece.

First, create a new web form by copying the webform_template.aspx and webform_template.aspx.py files from the chapter 11 source code. Rename these files to viewer.aspx and viewer.aspx.py respectively. Then edit viewer.aspx and change the `CodeFile` attribute in the `<%@ Page %>` directive at the top to point to viewer.aspx.py.[1]

Now that you have an empty web form, add a table with one row containing two cells to the page. You can do this either using the Design view, by dragging-and-dropping a table from the HTML section of the toolbox and deleting cells and rows as desired, or by typing the HTML in Source view. I (Christian) prefer the latter. The design view often generates messy HTML that's hard to work with later, and has problems round-tripping more complicated page layouts. It can even break your pages in some cases.

The right-hand side of the page is straightforward: it contains a heading and then two labels for the page title and page body respectively. Initially, the page will display the details of the first page in the document and, when a page link is clicked, will display the details of that page instead. Create these by either dragging two `Labels` into the right-hand table cell from the toolbox, or typing in the `asp:Label` tags. Give the `Label` controls the IDs `pageTitle` and `pageContent`, and add some formatting.

You'll create the list of links on the left-hand side of the page using a `Repeater` control. The `Repeater` is a container that repeats the controls in its *item template* once for each item in its data source list. In our case, you'll use the list of pages in the MultiDoc as the source, and the item template will contain a `LinkButton` control. `LinkButtons` are almost identical to `Buttons`, except that they're displayed as hyperlinks in the browser. The full ASPX code for the table is shown in listing 11.1.

---

[1] The process of copying and renaming template files and updating the new .aspx file to refer to the new codebehind is something that the IDE does automatically for supported languages, using the Add New Item… menu option. Language support for IronPython was included in an earlier release of the ASP.NET integration, but was not ported forward when the package was upgraded to work with IronPython 2. We hope it will be reinstated in the near future!

Listing 11.1 ASPX code for the MultiDoc Viewer user interface

```
<table>
  <tr valign="top">
    <td>
      <b>Pages:</b><br />
      <asp:Repeater id="pageRepeater" runat="server">
        <itemtemplate>
          <asp:LinkButton id="pageLink" runat="server" onclick="pageLink_Click"
       text="<%# title %>" enabled="<%# currentPage != title %>"/><br />
        </itemtemplate>
      </asp:Repeater>
    </td>
    <td>
      <b>Current page</b><br />
      <h2><asp:Label id="pageTitle" runat="server" /></h2>
      <asp:Label id="pageContent" runat="server" /><br />
    </td>
  </tr>
</table>
```

Look at the `asp:Repeater` element, which creates the `Repeater` control. It begins in a way similar to the web controls we've already created. The tag name is `asp:Repeater`, and you give it an id and specify that it runs on the server. Next, in the `<itemtemplate>` element, you specify the child controls that will be repeated. Inside this is the `LinkButton`, which has similar attributes to the button created in section 11.2.3—`id`, `runat`, `onclick`, and `text`—as well as a new attribute, `enabled`, which controls whether the `LinkButton` should respond to clicks. The values of the `text` and `enabled` attributes are also new—they're *data-binding expressions*.

DATA BINDING

`Repeaters` and other container controls can be filled using an ASP.NET feature called data binding—you give the control a data source (often in the code-behind), such as a list of items, and tell the control how to display each item (in the ASPX), and it does the rest. You can see the display code in listing 11.1; the code-behind side of the data binding looks like this:

```
pageRepeater.DataSource = document.pages
pageRepeater.DataBind()
```

When the `DataBind` method is called, the item template is duplicated for each item in the `DataSource`, and any data-binding expressions (snippets of code enclosed in `<%#` and `%>`) are evaluated. Note that these expressions are evaluated in the *context of the current item*. So in listing 11.1, the `<%# title %>` expression is evaluated in the context of a MultiDoc page object, and the text of the `LinkButton` is populated with the title of the page.

Data-binding expressions can be arbitrarily complex, and they can call methods or refer to global variables as well as the attributes of the current data item. The expression for the `LinkButton`'s `enabled` attribute, `<%# currentPage != title %>`, disables

the control when the title of the current data item matches the title of the page currently selected in the Viewer.

The clarity of these data-binding expressions is due, in part, to using IronPython. When using C# in ASP.NET, data-binding expressions are often much more complicated, due to the indirection required to express arbitrary attribute lookups in a statically typed language.

The ASPX code for the MultiDoc Viewer is complete; now you need to provide the behavior of the page in the code-behind file.

11.4.2 Code-behind

Open the code-behind file, viewer.aspx.py (if you're editing the ASPX page, you can hit F7), and add the code in listing 11.2.

Listing 11.2 Reading MultiDoc file and finding pages

```
from documentreader import DocumentReader
multidoc = None # the MultiDoc instance
currentPage = None # the title of the selected page

DOCUMENT_LOCATION = "doc.xml"
def getMultiDoc():
   reader = DocumentReader(Page.MapPath(DOCUMENT_LOCATION))
   return reader.read()

def getPage(multidoc, name):
   matches = [page for page in multidoc.pages if page.title == name]
   if matches:
      return matches[0]
   return None
```

This code provides the `DocumentReader` class, initializes the two pieces of state that the page will manage, and defines two simple functions; `getMultiDoc` uses the `DocumentReader` class to create a MultiDoc instance (using `MapPath` to avoid having to specify a full path to the doc.xml file), and `getPage` finds a page in the MultiDoc by its title (or `None` if there's no page with the given title).

The state variables are in the global scope; but, because of the way ASP.NET integrates the code-behind file, they aren't shared between requests (what you might expect if this were a normal Python module). In this case, storing the state in global variables is similar to the way the state would be managed in C#—they would be instance variables of the page subclass.

Now you need to hook into the ASP.NET page lifecycle to interact with the web controls on the page.

PAGE LIFECYCLE

A number of different events are raised for a page and its controls in the processing of a request. (See figure 11.1 for a more comprehensive list.) In this case, you need to handle the following:

- `Page_Load` *event*—Happens when the page is requested, after child controls have been created but before any handling of user input to the page has been done
- `Click` *events*—Are triggered when a user clicks on a `LinkButton` to select a page of the MultiDoc document
- `Page_PreRender` *event*—Is raised when the system is ready to convert the tree of web controls into HTML to be sent to the browser

Let's look at the code for each of these events in turn, beginning with `Page_Load` (listing 11.3).

Listing 11.3 MultiDoc Viewer `Page_Load` handler

```
def Page_Load(sender, event):
   global multidoc, currentPage
   multidoc = getMultiDoc()
   if not IsPostBack:
      currentPage = multidoc.pages[0].title
```

The `Page_Load` handler is automatically hooked up to the event by the ASP.NET machinery. In this listing, the code added to the handler loads the MultiDoc from the XML file; and, if this request isn't a postback (that is, it's caused by the user navigating to the page, rather than clicking a `LinkButton` on the page), it sets the currently selected page title to be the first in the document. (The global statement at the start of the function is required so that the assignments rebind the global variables, rather than shadowing them with local variables.) In listing 11.4, you can see the handler for clicking on page links.

Listing 11.4 MultiDoc Viewer `pageLink_Click` handler

```
def pageLink_Click(sender, event):
   global currentPage
   currentPage = sender.Text
```

The handler function defined in this listing is attached to each `LinkButton` created by the `Repeater` in the ASPX page. When a user clicks one of the page links, the `pageLink_Click` function will be called. Because the handler is attached to multiple `LinkButtons`, you use the `sender` parameter to work out the title that the user clicked, and store that title in the page state for when you render the page. The `PreRender` handler that uses the page state is shown in listing 11.5.

Listing 11.5 MultiDoc Viewer `Page_PreRender` handler

```
def Page_PreRender(sender, event):
   pageRepeater.DataSource = multidoc.pages
   pageRepeater.DataBind()
   selectedPage = getPage(multidoc, currentPage)
   pageTitle.Text = selectedPage.title
   pageContent.Text = selectedPage.text
```

The `PreRender` handler in this listing is triggered by ASP.NET when all postback event handling is completed, and allows you to use the page state that you've loaded and

modified in the other handlers to update the state of the web controls on the page. In the MultiDoc Viewer, you create the page links from the list of pages in the MultiDoc using data binding, and then you put the relevant parts of the selected page into the `pageTitle` and `pageContent` controls.

The MultiDoc Viewer is now complete. Hit the Play button, and the page should open in a browser window like figure 11.5.



Figure 11.5 The completed MultiDoc Viewer

You can click the links to view different pages. Try editing the XML file to add some extra pages in the document and see that they appear in the Viewer.

The structure of the code-behind may seem a little indirect at first glance: why not have the `pageLink_Click` handler set the page title and text itself, rather than setting internal page state and relying on the `Page_PreRender` handler to update the labels? This approach tends to cause duplication of code. You'd need to cater for the non-postback situation (when the `pageLink_Click` handler wouldn't be called), and populate the `pageTitle` and `pageContent` controls in the `Page_Load` handler. Additionally, in more complicated pages, more actions are available than clicking one of a list of links; keeping track of where you need to update which controls quickly becomes unwieldy. Writing the postback handlers (such as `Click` or `TextChanged`) to only update internal page state, and then having the `PreRender` handler translate that page state into the states of the child controls, is much more manageable when the possible interactions are wider.

You can see this clearly when we extend the MultiDoc Viewer to allow editing of the documents.

11.5 Editing MultiDocs

Let's extend the Viewer into an application that will enable you to update the Multi-Doc file, as well as looking at it. What do you need to add to the interface to support this? The simplest way is to add an Edit button to the page display area on the right side of the page. When the users click the Edit button, the page title and page

contents are swapped out with text boxes, allowing them to edit the values. The Edit button is replaced with a Cancel button and a Save button. Clicking either button will take the page out of edit mode, but the Save button also writes the updated content of the document out to the XML file.

These changes to the user interface allow you to edit a MultiDoc through the internet. Now we'll walk through making these changes, starting with the ASPX file.

11.5.1 Swapping controls

It's entirely possible to swap out the labels for editable text boxes by manipulating a web control's `Controls` collection. But it's often simpler and clearer to do so by hiding the controls you want to remove, and making some other (previously hidden) controls visible. A control that makes this technique especially convenient is the `Panel`, whose sole purpose is to contain other controls.

Let's update the ASPX page to do this. Change the right-hand cell of the table to contain the code found in listing 11.6.

Listing 11.6 Changing page display to allow showing and hiding

```
<b>Current page</b><br />
<asp:Panel id="viewPanel" runat="server" visible="false">
  <h2><asp:Label id="pageTitle" runat="server" /></h2>
  <asp:Label id="pageContent" runat="server" /><br />
  <asp:Button id="editButton" runat="server" text="Edit this page"
      onclick="editButton_Click" />
</asp:Panel>
```

The code in this listing wraps the `pageTitle` and `pageContent` labels in an `asp:Panel` element, which has its own ID and adds an Edit button after the labels. The panel will start off hidden (because `visible` is `false`). You can show the panel from the code-behind with the following code:

```
viewPanel.Visible = True
```

Now you add the `Panel` containing the controls that you want to display when the page is in edit mode (listing 11.7).

Listing 11.7 Controls to edit a MultiDoc page

```
<asp:Panel id="editPanel" runat="server" visible="false">
  <asp:TextBox id="pageTitleTextBox" runat="server" columns="40" /><br />
  <asp:TextBox id="pageContentTextBox" runat="server" columns="40" height="100"
      textmode="multiline" /><br />
  <asp:Button id="cancelButton" runat="server" text="Cancel"
      onclick="cancelButton_Click" />
  <asp:Button id="saveButton" runat="server" text="Save"
      onclick="saveButton_Click" />
</asp:Panel>
```

With the ASPX code in this listing added, the page now has two sets of controls that you can turn on and off, depending on whether it should be in reading mode or edit mode. The next step is to extend the code-behind page to handle the new controls.

We do need to think about one more thing, though. The original MultiDoc Viewer only has one piece of state that changes: the current page title. When someone clicked a page link and changed that state, you were always given the new value for it (as the text of the `LinkButton` that was the `sender` argument to the handler), so you could set up the page correctly. In the MultiDoc Editor, there are now two different pieces of state that can be changed: the current page title, and whether the page is in edit mode. When you receive an Edit button `Click` event, you aren't also told what the current page should be. Correspondingly, if you receive a page link `Click`, you don't know whether the page should be in edit mode. You need some other way to persist this information from the handling of one page request to the next.

The facility that ASP.NET provides to deal with this problem is called *view state*.

11.5.2 Handling view state

View state is an important concept in ASP.NET. At the end of processing a request, the state of all the controls on the page is serialized and stored in a hidden form field in the HTML page sent back to the client. When the next request from the client is received, serialized state is reconstituted and set back into the controls before any postback events are raised. (To ensure that the client hasn't monkeyed with the state it sends back, the framework cryptographically signs the view state and validates it before deserializing it.)

View state is managed automatically for web controls (unless it has been turned off for the control by setting `EnableViewState` to `false`), but you need to add some extra state to the process. In a C# web project, you could do this by overriding the `SaveViewState` and `LoadViewState` methods to inject the extra state to be saved. Unfortunately, because in IronPython you don't directly inherit from the ASP.NET `Page` class, the `SaveViewState` and `LoadViewState` methods don't get called in IronPython pages. To hook into the view state system, you need a little C# to call custom methods in the code. You put a `CustomScriptPage` class into the `App_Code folder` in the web project (listing 11.8).

Listing 11.8 `CustomScriptPage` class for delegating view state handling to Python

```
using Microsoft.Web.Scripting.UI;
using Microsoft.Web.Scripting.Util;

public class CustomScriptPage: ScriptPage {
   protected override void LoadViewState(object savedState) {
      DynamicFunction f = this.ScriptTemplateControl
      ➥  .GetFunction("ScriptLoadViewState");
      if (f == null) {
         base.LoadViewState(savedState);
      } else {object baseState = this.ScriptTemplateControl
      ➥  .CallFunction(f, savedState);
         base.LoadViewState(baseState);
      }
   }

   protected override object SaveViewState() {
      DynamicFunction f =this.ScriptTemplateControl
      ➥  .GetFunction("ScriptSaveViewState");
```

```
      if (f == null) {
         return base.SaveViewState();
      } else {
         object baseState = base.SaveViewState();
         return this.ScriptTemplateControl.CallFunction(f, baseState);
      }
   }
}
```

This listing creates a subclass of `ScriptPage` (the base class of all IronPython ASP.NET pages) that will look up functions called `ScriptLoadViewState` and `ScriptSaveViewState` in the script of the page (the Python code, in this case), and delegate to them. You can then tweak the view state objects in the code-behind. To declare that our page should inherit from this class, you need to change the first line of the ASPX page to add the `Inherits` option.

```
<%@ Page Language="IronPython" CodeFile="Default.aspx.py"
Inherits="CustomScriptPage" %>
```

To simplify much of the state management in the page, you can use a minimal page state class to group the pieces of state together into one object. This lets you leave out most of the global statements in event handlers, and makes it easier to work out where the names being used in a section of code are coming from. As you can see in listing 11.9, we've decided to call the object containing the page state `self`, which is a little non-standard, but seems to have the right feel in code using the state. (If this seems wrong, you can happily call it `state` instead.)

Listing 11.9 A class to group together page state

```
class PageState(object):
   pass
self = PageState()
self.document = None
self.currentPage = None
self.editing = False
```

Once you have that, you can save and load the page state (listing 11.10).

Listing 11.10 Loading and saving the MultiDoc Editor view state

```
from System.Web.UI import Pair
import pickle

def ScriptSaveViewState(baseState):
   state = Pair()
   state.First = baseState
   state.Second = pickle.dumps((self.document, self.currentPage,
     self.editing))
   return state

def ScriptLoadViewState(state):
   self.document, self.currentPage, self.editing =
➥     pickle.loads(state.Second)
   return state.First
```

The ASP.NET view state machinery can't natively serialize Python types, so in this listing the `ScriptSaveViewState` function creates a `Pair`. A `Pair` is a .NET class, a clumsy version of Python's tuple, which can contain two other objects (as `.First` and `.Second`) and can be serialized in the view state. You store the standard page view state in `.First`, and use the Python serialization module `pickle` to create a string from a tuple of the three pieces of state you need to preserve. Then, in `ScriptLoadViewState`, you receive the view state object that ASP.NET has pulled out of the request, deserialize our custom page state tuple from the `pair.Second`, and return the other half back to the machinery to restore the child control state.

You can piggyback almost anything in the view state in this way, as long as you ensure that the original view state is maintained and that anything you add can be serialized by ASP.NET. (The `pickle` module is very useful here.) Also, you need to be careful that the functions loading and saving the view state are symmetrical—any change in the way you store your custom state needs to be reflected in how it's loaded again. One more thing to keep in mind is that the view state is sent to the client in each response, and then back to the server with the subsequent request, so it can't be used to store large volumes of data.

As well as code for loading and saving the view state of the page, you need to include the code for loading and saving MultiDoc files (listing 11.11).

Listing 11.11 Loading and saving MultiDoc file

```
from documentreader import DocumentReader
from documentwriter import DocumentWriter

DOCUMENT_FILE = "doc.xml"

def getDocument():
   reader = DocumentReader(Page.MapPath(DOCUMENT_FILE))
   return reader.read()

def saveDocument(document):
   writer = DocumentWriter(Page.MapPath(DOCUMENT_FILE))
   writer.write(document)

def getPage(document, name):
   matches = [page for page in document.pages if page.title == name]
   if matches:
      return matches[0]
   return None

def Page_Load(sender, event):
   if not IsPostBack:
      self.document = getDocument()
      self.currentPage = None
      self.editing = False
```

This listing shows the familiar `getDocument` and `getPage` functions and adds the new `saveDocument` function. It also includes the `Page_Load` event that determines how the state is initialized in the first request, when `IsPostBack` is `false` and no view state is processed.

Because you're now storing the page state in a wrapper object named `self`, you need to change the data-binding expression in the ASPX file from `currentPage != title` to `self.currentPage != title`.

Next you handle the `Page_PreRender` event (to display the state of the page), and the postback events for the controls you've added: `Click` events from the page links; the Edit, Save, and Cancel buttons; and the `TextChanged` events from the title and body text boxes.

11.5.3 Additional events

To start with, let's consider how our page's state should be displayed in the web controls in the `Page_PreRender` handler (listing 11.12). This handler will be called after all of the click-type postback events in the page's lifecycle.

Listing 11.12 MultiDoc Editor `Page_PreRender` handler

```
def Page_PreRender(sender, event):
   pageRepeater.DataSource = self.document.pages
   pageRepeater.DataBind()
   viewPanel.Visible = self.currentPage and not self.editing
   editPanel.Visible = self.editing

   if self.currentPage:
      selectedPage = getPage(self.document, self.currentPage)
      pageTitle.Text = pageTitleTextBox.Text = selectedPage.title
      pageContent.Text = pageContentTextBox.Text = selectedPage.text
```

In this listing, the `Page_PreRender` handler databinds the `pageRepeater` in the same way as the previous version of the handler (listing 11.5), creating the page links on the left-hand side of the page. Then you decide which (if either) of the right-hand side panels should be displayed. Finally, if you have a current page, you populate the labels and text boxes for the view and edit panels.

The other event handlers, in listing 11.13, are simple—they merely record the user input in the page state, and rely on the `PreRender` handler to display the page in a consistent way, whereas the view state is managed by the `Save` and `Load` functions. Using the `PageState` holder class means that the handlers don't need any global statements cluttering up the code.

Listing 11.13 Postback event handlers in MultiDoc Editor

```
def pageLink_Click(sender, event):
   self.currentPage = sender.Text

def editButton_Click(sender, event):
   self.editing = True

def cancelButton_Click(sender, event):
   self.editing = False
   # throw away any changes that have been made
   self.document = getDocument()

def saveButton_Click(sender, event):
```

```
    saveDocument(self.document)
    self.editing = False

def pageTitleTextBox_TextChanged(sender, event):
    selectedPage = getPage(self.document, self.currentPage)
    selectedPage.title = self.currentPage = pageTitleTextBox.Text

def pageContentTextBox_TextChanged(sender, event):
    selectedPage = getPage(self.document, self.currentPage)
    selectedPage.text = pageContentTextBox.Text
```

In this listing, the `TextChanged` event handlers for the page title and page body are interesting because typing into the text box doesn't trigger a postback by default (although it can if the text box has `AutoPostBack` set to `True`). In this case, when ASP.NET restores the view state and then applies the posted data to the controls, it detects that the text has changed and raises the event. You use this notification to update the document before the Save button or page link `Click` events are raised.

Try running the MultiDoc Editor now. You can view the different pages and click the Edit button to change page titles and bodies. When you click the Save button, the file is updated on the disk.

Beware, though! If you enter anything that looks like HTML tags, you'll see a rather alarming error page, beginning with the message: *A potentially dangerous Request.Form value was detected from the client.* This is a security feature in ASP.NET intended to prevent cross-site scripting (XSS) attacks, but it's so heavy-handed that the idea must be to shock developers into crafting a strategy to handle HTML in user input, after turning the validation off. Obviously, how you prevent cross-site scripting depends on the extent to which you trust the user—in this case, it's you. To turn off validation, edit the Web.config file, and add `validateRequest="false"` to the `<pages/>` element.

At this point, the MultiDoc Editor is largely complete. The only significant feature missing is a New Page button; adding this is straightforward with the techniques used so far, so we don't cover it here. Another consideration that we haven't examined is how to handle two people editing the document at the same time. Obviously, the second person to save changes will overwrite those of the first. You can handle this in many ways—for example, using a timestamp on the document or a locking protocol. A robust solution might involve a database; we discuss using the .NET facilities to communicate with databases in the next chapter.

You can see an example of what the Editor looks like in figure 11.6.

Now that you have a MultiDoc Editor, you might want to use it within a larger website. In ASP.NET, packaging up the ASPX and Python code is done by creating a *user control*. The control can then be embedded into ASPX pages or other user controls, in almost the same way as you'd incorporate standard web controls. Let's look at what changes you need to make the Editor reusable.

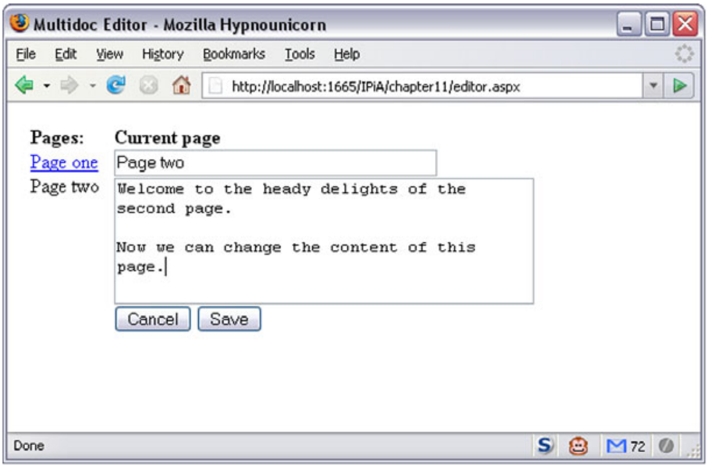


Figure 11.6 Editing a MultiDoc page in a browser

11.6 Converting the Editor into a user control

First, create an empty user control to hold the code of our page, by copying the user control template files from the chapter 11 code into the web project. Copy both usercontrol_template.ascx and usercontrol_template.ascx.py to the folder, and rename them as MultiDocEditor.ascx and MultiDocEditor.ascx.py. Open up the copied .ascx file, and you'll see that it starts with the following line:

```
<%@ Control Language="IronPython" CodeFile="usercontrol_template.ascx.py" %>
```

This is almost exactly the same as the page directive at the start of an ASPX page, except for the word `Control`, and the extension of the code-behind file. The `Control` directive is all there is in this file, though; it doesn't have the HTML (the `doctype`, as well as `html`, `head`, `body`, and `form` elements) that empty ASPX pages have. User controls are never rendered directly to web clients; they're always contained in an ASPX page that wraps them (and any other controls in the page) with the normal HTML structure. Other than that difference, writing a user control is similar to writing a normal ASPX web page.

Change the `Control` directive so that the `CodeFile` attribute refers to MultiDocEditor.ascx.py.

Now create a new web form called Container.aspx by copying the template web forms, renaming them, and editing the `CodeFile` attribute in the .aspx. This will be the page that wraps the user control. After the page directive, add the following line:

```
<%@ Register src="MultiDocEditor.ascx" tagname="multidoceditor"
tagprefix="ipia" %>
```

This `Register` directive allows you to include the `MultiDocEditor` user control in the web page. You add the following line inside the `<form>` element of the page:

```
<ipia:multidoceditor id="editor" runat="server" />
```

You can see that the element name has been constructed from the `tagprefix` and `tagname` parameters in the register directive. The tag prefix (`ipia` in the example) can be anything you like; this can be useful when you want to disambiguate controls from different sources that have the same name.

Now the container page embeds the editor control, although the control doesn't do anything yet. You could put some text into the ASCX file and view Container.aspx in the browser, and the page would display that text where you've added the control. Then you populate the control with the code you already have by doing the following:

- Copying the table from the MultiDoc Editor ASPX page into the ASCX
- Copying all the code from the original code-behind file to the new one

Unfortunately, before the control will work, we need to revisit the view state handling.

11.6.1 View state again

When implementing the Editor as a web page, our page inherited from a custom subclass of `ScriptPage`. User controls can also inherit from a custom base class, but it needs to be a subclass of `ScriptUserControl`. You can create a custom subclass to forward the view state handling methods with the code in listing 11.14.

Listing 11.14 Delegating view state handling to Python

```
using Microsoft.Web.Scripting.UI;
using Microsoft.Web.Scripting.Util;

public class CustomScriptUserControl: ScriptUserControl {
   protected override void LoadViewState(object savedState) {
      ScriptTemplateControl stc = (this as
     IScriptTemplateControl).ScriptTemplateControl;
      DynamicFunction f = stc.GetFunction("ScriptLoadViewState");
      if (f == null) {
         base.LoadViewState(savedState);
      } else {
         object baseState = stc.CallFunction(f, savedState);
         base.LoadViewState(baseState);
      }
   }

   protected override object SaveViewState() {
      ScriptTemplateControl stc = (this as
     IScriptTemplateControl).ScriptTemplateControl;
      DynamicFunction f = stc.GetFunction("ScriptSaveViewState");
      if (f == null) {
         return base.SaveViewState();
      } else {
         object baseState = base.SaveViewState();
         return stc.CallFunction(f, baseState);
      }
   }
}
```

This code is almost the same as the CustomScriptPage definition in listing 11.8, with one odd difference—for some reason, ScriptUserControl doesn't expose a .ScriptTemplateControl directly. Instead, it explicitly implements the property on the IScriptTemplateControl interface; the definition of the property looks something like this:

```
public virtual ScriptTemplateControl
IScriptTemplateControl.ScriptTemplateControl { get {…} }
```

You need to cast this ScriptUserControl to an IScriptTemplateControl before you can get hold of the .ScriptTemplateControl property. Ordinarily, you'd use explicit interface implementation to disambiguate when inheriting from two interfaces that expose the same member and you'd need to provide different implementations, but that doesn't seem to be the situation here.

At this point in the section, you could be forgiven for thinking that wrapping up a page into a user control is rather a lot of work. Forwarding the view state between the C# and Python code of a control is definitely one part of the ASP.NET–IronPython integration that's still fairly nasty, and it's something that we expect will be fixed in future versions as the IronPython support matures. In any case, the classes you have work well now, so you can simply reuse the classes when you need to manage view state from IronPython pages and user controls, without worrying too much about the fact that they're workarounds for gaps in the system.

Once the CustomScriptUserControl class is in App_Code, you can change the user control to inherit from it by adding Inherits="CustomScriptUserControl" to the Control directive in MultiDocEditor.ascx. The view state handling in the code-behind file will now be called by the ASP.NET machinery, and the MultiDocEditor user control will work! The only problem is that it's not reusable except in a very basic sense: you can put it on different pages, or several instances on one page, but each instance will be editing the same file. That isn't especially useful. You need to be able to specify which file a particular editor should be using. You can do this by adding a filename parameter to the control.

11.6.2 Adding parameters

If you were implementing the user control in C#, you'd expose a filename parameter by creating a property on the class. In IronPython, you make the filename settable in essentially the same way, although it doesn't use the Python property system (because the IronPython code-behind file doesn't directly define a class). Add the functions in listing 11.15 to the MultiDocEditor code-behind file to give the user control a Filename property.

Listing 11.15 Creating the Filename property

```
def GetFilename():
   return self.filename

def SetFilename(filename):
   self.filename = filename
```

These functions will be called by the ASP.NET infrastructure when you specify the filename in ASPX pages that embed the control. (The case of the attribute needs to match the name of the getter and setter functions.)

```
<ipia:multidoceditor id="editor" runat="server" Filename="doc.xml" />
```

Alternatively, the getter and setter functions can be called from the code of the container page.

```
def Page_Load(sender, e):
   editor.SetFilename("doc.xml")
```

To complete the process, the code-behind needs a few changes to make use of the filename that's passed in. First, the filename needs to be added to the page state.

```
self = PageState()
self.filename = None
self.document = None
self.currentPage = None
self.editing = False
```

Then it needs to be loaded and saved with the view state.

```
def ScriptLoadViewState(state):
   self.filename, self.document, self.currentPage, self.editing =
➥     pickle.loads(state.Second)
   return state.First

def ScriptSaveViewState(baseState):
   state = Pair()
   state.First = baseState
   state.Second = pickle.dumps((self.filename, self.document,
     self.currentPage, self.editing))
   return state
```

And of course it should be used to load and save the MultiDoc file:

```
def getDocument():
   reader = DocumentReader(Page.MapPath(self.filename))
   return reader.read()

def saveDocument(document):
   writer = DocumentWriter(Page.MapPath(self.filename))
   writer.write(document)
```

With these changes, the MultiDoc Editor user control is complete! You can view the container page in the browser and edit the document as you could in the case of the single web page. You can add another `<ipia:multidoceditor/>` element to the container page, specify a different MultiDoc file, and edit the two documents independently in one page. Or you can even create another user control that wraps the `MultiDocEditor` control—perhaps one that lists the MultiDoc files in a directory in a drop-down list, and allows the user to select which file to edit. As you can see, user controls provide a flexible mechanism for packaging and reusing components when constructing a web application.

11.7 Summary

This chapter has been a look at the basics of creating ASP.NET web pages and user controls with IronPython. You've created a web application, and a web page to display MultiDoc documents. Then you extended the web page with more complex interactions to allow you to edit the documents. Finally, you packaged this functionality into a reusable user control that can be embedded into web pages and other controls. Along the way we've touched on some of the support ASP.NET provides for building applications, including view state management, debugging, configuration, and the special App_Script and App_Code directories for shared code. There's a lot more in the framework; in fact, there are whole books devoted to it! We hope that our overview has given you enough information on the underpinnings of the system that you can take other resources written with C# or Visual Basic in mind and apply them to IronPython.

The support for IronPython in ASP.NET is still new, and there are some rough edges—notably in the view state handling where you needed to use C# to forward calls to IronPython. The IronPython–ASP.NET integration is a work in progress. There are big plans for the future, so we can look forward to the holes being filled soon.

In the next chapter, we look at using databases and web services with IronPython.

12 Databases and web services

This chapter covers

- Using ADO.NET to work with databases
- Interacting with SOAP and REST web services
- Creating a REST web service

So far in the book we've concentrated on creating fairly self-contained applications; most data storage has been in the form of text or XML files. For a lot of tasks this can be sufficient, but often the data your application will be manipulating lives somewhere else: maybe in a relational database or behind a web service. One application I (Christian) worked on was part of a flight-planning system; weather maps and forecasts that the application needed were managed by another system accessible through a separate web service, while user preferences and details were stored in a local SQL Server database.

In this chapter we look at some of the techniques we can use to get access to this data. To start with, we cover the base API that the .NET framework provides for talking to different relational databases, followed by the higher-level classes that can be

layered on top for slicing and dicing data in your application. Then we see how we can interact with different types of web services from IronPython.

12.1 Relational databases and ADO.NET

Relational database management systems (RDBMSs) are everywhere in the IT industry, and there is a huge range of different database engines, from open source projects such as MySQL and PostgreSQL to commercial vendors like Microsoft SQL Server and Oracle. While the various engines often have quite different capabilities and feature sets, they all use a common model to represent data (the *relational model*) and provide a standardized[1] language for getting that data: SQL (Structured Query Language). However, although the language is standard, the methods for connecting to a database and sending an SQL query to it can often be quite different. ADO.NET (the .NET Database Communication layer) is designed to solve this problem.

What is the relational model?

It's a formal framework for reasoning about databases, devised by Edgar Codd in 1969. It defines databases in terms of *relations*, which are sets of things that Codd called *tuples*, although from a Python perspective they're more like dictionaries keyed by strings. All of the tuples in a relation have the same *attributes* (which are the keys of the dictionaries), and the values of the attributes have to be *atomic*: they can't be lists or tuples themselves. In relational database management systems, relations are called tables, tuples are rows, and attributes are columns.

From that foundation, the relational model expands to define *keys* (sets of attributes that uniquely identify a tuple within a relation) and a language for expressing queries called the *relational algebra* (which is what SQL is based on).

You can find a lot more information about the relational model on Wikipedia.

In section 12.1, we cover the basics needed to use the .NET communication components in IronPython. We start by looking at the structure of the ADO.NET layers, and then we explore the methods they provide for managing relational data. By the end of the section, you should have a good understanding of the basics of talking to any database from IronPython. You'll also have enough grounding to use examples written for other .NET languages to learn more advanced ADO.NET techniques.

So what is ADO.NET? The *ADO* part comes from the previous Microsoft approach to data access, ActiveX Data Objects, but there's really only a slight spiritual relationship between ADO and ADO.NET. It's tricky to describe precisely, because ADO.NET isn't a class library itself. Instead, it's a design for a large suite of related data access libraries, with a layer of classes that you can use to provide uniform data manipulation on top, as Figure 12.1 shows.

[1] Well, standardized to an extent. There are large differences between the SQL supported in different databases, but generally the core operations (select, insert, update, and delete) work the same.

The bottom layer of ADO.NET is the Data Provider layer. Each DBMS has its own ADO.NET data provider, which wraps the specific details for connecting to and interacting with that database in a collection of classes with a common interface. You can see the most important classes in a data provider, as well as their roles, in table 12.1. We'll look at these in more detail soon.

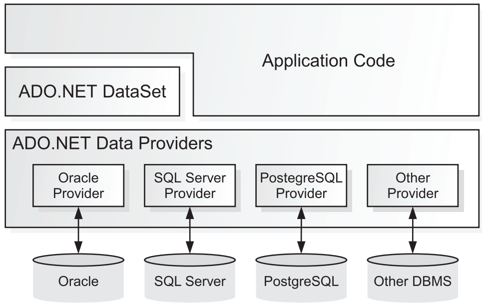


Figure 12.1 Application code can talk to databases using data providers directly or via `DataSets`.

Table 12.1 The core classes in a data provider

Class	Description
`Connection`	Maintains the channel of communication with the database engine
`Command`	Allows executing commands to query and altering data in the database through a connection
`DataReader`	Gives access to the stream of records resulting from a query
`DataAdapter`	Manages the movement of data between the Data Provider layer and the layer above it: the ADO.NET `DataSet`

The layer above the data providers contains the `DataSet` and its component classes. `DataSets` are used to present and manipulate data pulled in from external sources in memory and to save changes made to the data back to the database (or another data store). A `DataSet` is a container for relational data and can contain a number of `DataTables` (each of which represents a table in the database) as well as relationships between the tables, which can be used to navigate the data and maintain consistency. You can see in figure 12.2 how the classes fit together.

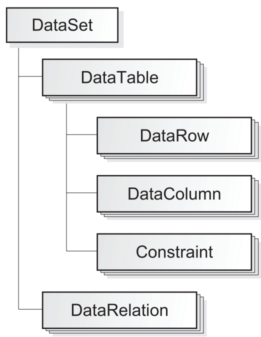


Figure 12.2 The structure of a `DataSet`

So that's the basic structure of ADO.NET data providers and `DataSets`. In the rest of this section, we look at using them from IronPython. We start with the data provider class-es: connecting to the database, issuing commands, querying the data, and using transactions to bundle changes together. Then we see how `DataSets` fit on top, using the `DataAdapter`.

All of this is quite abstract. Let's see it in action (and in more detail).

12.1.1 Trying it out using PostgreSQL

To begin exploring the Data Provider layer, we need a database to work with and a corresponding data provider to connect to it from IronPython. In these examples we're going to use PostgreSQL, a high-quality open source DBMS. You can download the

database engine from www.postgresql.org, and the ADO.NET data provider for PostgreSQL is Npgsql, which is available at npgsql.org. You can administer Postgres databases completely through the command-line tools that come with it, but if you'd like a GUI administration tool to manage the database, PGAdmin (available from www.pgadmin.org or included with recent versions of PostgreSQL) works well.

Finding out more about PostgreSQL and Npgsql

PostgreSQL has an extensive manual available from http://www.postgresql.org/docs/manuals/.

The main tools for working with PostgreSQL databases are the psql command-line client and PGAdmin. You can run both of them from the Start menu. When using psql, enter `\?` for help on psql commands and `\h` for help on SQL commands.

There's a whole host of information about advanced uses of Npgsql in the user manual at http://npgsql.projects.postgresql.org/docs/manual/UserManual.html.

Using another DBMS with the examples in this chapter

ThisWhile the database examples in this chapter have been written with PostgreSQL in mind, it should be straightforward to follow along with a different database system. You'll need to change a few things:

- The script that creates the example database uses PostgreSQL-specific commands to create the tables, sequences, and foreign key constraints. You should change the creation script to use the syntax for your database or create the tables yourself through the administration interface. The insert statements that populate the database tables should work as is.
- Rather than installing Npgsql, you'll need to download and install the ADO.NET data provider for your database, if you haven't already.
- Where we add a reference to the Npgsql assembly and import from Npgsql, check the provider's documentation to find out the assembly you need to reference and the namespace that contains your provider's classes. Where the examples use `NpgsqlConnection` or `NpgsqlDataAdapter`, you should use the corresponding classes in your data provider.
- When you create a database connection, the details of the connection string will be slightly different. Again, the provider's documentation will specify which parameters are expected.

With those changes in place, you can use any database engine to work through the examples.

INSTALLING THE PIECES

Download and install PostgreSQL. For our purposes, the default settings will work well. Then run the SQL script chapter12\12.1.1\schema.sql from the source code for the book using the following command:

```
psql -U postgres -f schema.sql
```

If psql isn't on your path, you'll need to include the full path to the PostgreSQL bin directory. This will connect to the database server as the postgres user and create a new database called ironpython_in_action, with tables you can use for experimenting with ADO.NET in IronPython. The database contains information about movies, actors, and directors; figure 12.3 shows the fields in the tables and the relationships between them.

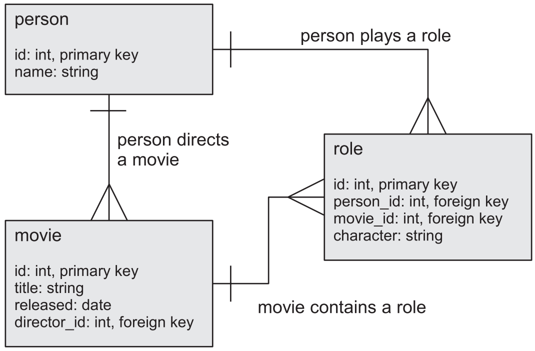


Figure 12.3 The example database stores information on movies, people, and roles and how they're related.

The final new piece that you need is the PostgreSQL data provider, Npgsql. Download the latest binary distribution (at the time of writing this was 2.0.2) for your platform, and unzip it into a directory. To ensure that you can use it from IronPython, add the directory containing Npgsql.dll to your IRONPYTHONPATH environment variable. Once that's all done, you should be able to import Npgsql and instantiate the classes it provides to communicate with the database.

12.1.2 Connecting to the database

One of the things I like about Python is that I can explore a new API in the interactive interpreter and get immediate feedback, rather than having to build a test program and then realizing that I've misunderstood how the API works. So let's explore: run ipy, and enter the following commands:

```
>>> import clr
>>> clr.AddReference('Npgsql')
>>> import Npgsql as pgsql
```

If the last import succeeds, then the interpreter has found and loaded the Npgsql assembly, and we can try to connect to the database. To do this, we use the NpgsqlConnection class (each data provider defines its own connection class).

```
>>> connection = pgsql.NpgsqlConnection(
... 'server=localhost; database=ironpython_in_action;'
... 'user id=postgres; password=<postgres user password>')
```
[2]

(The password parameter should be the postgres user's password that you specified when installing PostgreSQL.)

The parameter the NpgsqlConnection constructor takes is a *connection string* specifying the details of the database we're connecting to. What it should contain and how it is formatted depends on the data provider, although all the providers I've used have had the same format you see here. Also, while our connection string is pretty close to minimal, there's a great variety of optional parameters that you can specify in

[2] Note that there is no comma between these two strings. We're making use of the fact that Python will join adjacent string literals together to fit this connection string on the page nicely; you can enter it as one long string if you're typing it.

the connection string, such as whether connections should be pooled, whether the communication should be encrypted with SSL, and so on. To determine exactly what parameters a specific provider expects, you need to look in the documentation for that provider; another useful reference for a large number of databases is at www.connectionstrings.com.

```
>>> connection.Open()
```

The `Open` method does what it says on the tin: it creates the underlying communication channel to the database and logs in. If there were any network problems stopping us from reaching the database, or we'd given incorrect credentials in the connection string, the `Open` call would fail with a (hopefully) helpful error message.

In an application (particularly a web application) it's very important to ensure that connections that are opened are closed (by calling the `Close` method) again. Database connections are external resources that generally are not totally managed by the .NET framework, and often they're pooled to reduce the expense of creating and opening them. In a pooling situation, calling `Close` will return the connection to the pool for another task to use.

Before closing this connection, though, let's look at how we can use it to have the database do our bidding.

12.1.3 Executing commands

Using our connection we can execute commands against the database; essentially, anything we could do using `psql` (the PostgreSQL command shell), we can do now from Python code.[3] To insert a row into the person table, we would execute the code in listing 12.1.

Listing 12.1 Inserting a record

```
>>> command = connection.CreateCommand()
>>> command
<Npgsql.NpgsqlCommand object at 0x... [Npgsql.NpgsqlCommand]>
>>> command.CommandText = "insert into person (name) values ('Danny Boyle')"
>>> command.ExecuteNonQuery()
1
```

We call the `Connection.CreateCommand` method to get an `NpgsqlCommand` associated with the connection. Then we set the command text to a SQL insert statement and call the command's `ExecuteNonQuery` method, which runs the statement and returns the number of records affected by the statement. The `NonQuery` in the method name indicates that we don't expect any data back in response.

Imagine we have a list of actors that we had loaded from somewhere else and want to insert into the person table. We could use a loop to put the people into the database, constructing the insert statement using `%` string formatting, as shown in listing 12.2.

[3] Admittedly, it's a bit more verbose, but there are also definite benefits, like being able to run insert statements in loops, generate SQL statements programmatically, or save the result of a query in a variable for future use.

Listing 12.2 Trying to insert multiple people by constructing SQL strings

```
>>> people = ['James Nesbitt', "Sydney 'Big Dawg' Colston",
... 'Helena Bonham Carter']
>>> insert_statement = "insert into person (name) values ('%s')"
>>> for p in people:
...    command.CommandText = insert_statement % p
...    command.ExecuteNonQuery()
Traceback (most recent call last):
  File , line unknown, in Initialize##365
  File Npgsql, line unknown, in ExecuteNonQuery
  File Npgsql, line unknown, in ExecuteCommand
  File Npgsql, line unknown, in CheckErrors
EnvironmentError: ERROR: 42601: syntax error at or near "Big"
```

Oops—in listing 12.2, the single quotes in Sydney Colston's name have been interpreted as the end of the string in the command, and the next part of his name isn't valid SQL, so the database has thrown a syntax error. We could avoid this by replacing every single quote with two single quotes (this is the SQL method of embedding quotes in strings), but that's easy to forget and error prone. ADO.NET provides another way to handle SQL commands with values that vary: command parameters. (These are sometimes called *bind variables* in other database APIs.)

SQL injection

Actually, the fact that "Big Dawg" isn't valid SQL is quite lucky: if his nickname had been "); DROP TABLE movie CASCADE; --"[4] we might have lost a big chunk of our data. This would be a huge security flaw in a web application: it's an attack called *SQL injection*. Even if the permissions on the database tables have been locked down so that the attacker couldn't drop a table, the attacker might be able to log in with elevated privileges or view or edit other users' information. Command parameters are the solution to this problem.

In general, if you find yourself constructing SQL strings programmatically, you should think twice. At the very least, check to see whether what you are doing can be done using parameters instead.

12.1.4 Setting parameters

To use a command with parameters, we set its `CommandText` to be a string containing references to parameters, rather than literal values:

```
>>> command = connection.CreateCommand()
>>> command.CommandText = 'insert into person (name) values (:name)'
```

[4] This command will delete the movie table with all of its data and all of the relationships with other tables. The extra characters are to ensure that the resulting string is valid SQL when concatenated with the text of the rest of the command in an insecure way.

This is similar to the `insert_statement` variable we used earlier, except that it uses `:name` instead of `%d` to refer to the value we want. Also, the single quotes are not needed, because the parameter value is never stitched into the string. It's passed in separately, and the database engine uses it as it executes the statement. This is what makes parameterized commands safer and easier to program with: you don't need to worry about quoting strings, and for dates you can simply pass the date object rather than having to convert them into a string in a format the database engine will understand.

Now our loop to insert multiple people looks like the code in listing 12.3.

Listing 12.3 Setting parameter values in a parameterized command

```
>>> command.Parameters.Add('name', None)
>>> for p in people:
...     command.Parameters['name'].Value = p
...     command.ExecuteNonQuery()
```

Another benefit of using parameters is that they are more efficient if you are executing a statement more than once with different parameters. Parsing an SQL statement is expensive, so databases will usually cache parsed form and reuse it if exactly the same statement is sent again. Parameterized commands let you take advantage of that behavior.

Now we can send commands to the database to insert, update, or delete rows. Next we look at the methods that allow us to get information back.

12.1.5 Querying the database

The simplest form of query is one that returns a single value, for example, retrieving an attribute of an item in the database or counting the records that match search criteria. In that situation, we can use the `ExecuteScalar` method of the command, which you can see in listing 12.4.

Listing 12.4 Getting a value from the database with `ExecuteScalar`

```
>>> command = connection.CreateCommand()
>>> command.CommandText = 'select count(*) from person'
>>> command.ExecuteScalar()
5L
>>> command.CommandText = 'select name from person where id = 1'
>>> command.ExecuteScalar()
'David Fincher'
```

As you can see, `ExecuteScalar` returns whatever type of object the query returns; this can make it a bit of a hassle in a statically typed language like C#, where you'd have to cast the result before you could use it. In Python, this kind of function is convenient.

A minor point to note is the way `ExecuteScalar` deals with `NULL`[5] values in the database or queries that return no rows. Both of these are displayed in listing 12.5.

[5] `NULL` is a special SQL value used in databases to indicate that no value has been specified in a column. In a way it's similar to Python's `None`, but there are a number of differences in the way `NULL`s behave when used in arithmetic or Boolean expressions with other values.

Listing 12.5 `ExecuteScalar` with `NULL` or no rows

```
>>> from System import DBNull
>>> command.CommandText = 'select null'
>>> command.ExecuteScalar() is DBNull.Value
True
>>> command.CommandText = 'select id from person where 1 = 2'
>>> command.ExecuteScalar() is None
True
```

When the query in listing 12.5 returns a row with a `NULL` in it, `ExecuteScalar` gives us `DBNull`, a special singleton value in .NET that is similar to `None`. When there are no rows returned, `ExecuteScalar` gives us `None` instead.

We can also call `ExecuteScalar` with a query that returns multiple rows:

```
>>> command.CommandText = 'select id, name from person'
>>> command.ExecuteScalar()
1L
```

In this case, it returns the first value of the first row of the results. If we want to see more of the results, we'll need to use a `DataReader`.

12.1.6 Reading multirow results

To get the results from a query, we call `ExecuteReader` on a command (using the same command text as above):

```
>>> reader = command.ExecuteReader()
>>> reader
<NpgsqlDataReader object at 0x…>
```

Now the `DataReader` is ready to start reading through the rows. Readers have a number of properties and methods for examining the shape of the rows that have come back. You can see some of these used in listing 12.6.

Listing 12.6 Finding out what data is in a `DataReader`

```
>>> reader.HasRows
True
>>> reader.FieldCount
2
>>> [(reader.GetName(i), reader.GetFieldType(i))
...  for i in range(reader.FieldCount)]
[('id', <System.RuntimeType object at 0x...[System.Int32]>),
('name', <System.RuntimeType object at 0x... [System.String]>)]
```

We use the `HasRows` and `FieldCount` properties and the `GetName` and `GetFieldType` methods to see the structure of the data returned for our query.

Readers also have a current row; when the reader is opened, it's positioned before the first row in the results. You advance to the next row with `Read`, which returns `True` unless you've run out of rows. Getting the values of the fields in the current row can be done in a number of ways. The most convenient is by indexing

into the reader by column number or name. Putting this together, we have the loop shown in listing 12.7.

Listing 12.7 Getting values from the current row

```
>>> while reader.Read():
...   print reader['id'], reader['name']
1 David Fincher
2 Edward Norton
3 Brad Pitt
# and so on...
```

A number of other methods are available for getting fields of the current row, such as `GetDecimal`, `GetString`, and `GetInt32` (and nine more!), but they're not especially useful in Python. Their only purpose is to inform the compiler what type you want (to avoid a cast), but they'll fail (at runtime) if you request a field as the wrong type. Worse, they accept only an integer column index, so they'll break if the query changes to reorder the columns, unless you use the `GetOrdinal` method of the reader to look up the name first. In a choice between `reader.GetString(reader.GetOrdinal('name'))` and `reader['name']`, I vastly prefer the latter.

Some databases will allow you to make several queries in one command, and then they return multiple result sets attached to one reader. In this case you can use the `NextResult` method to move to the next set of rows. Similarly to `Read`, `NextResult` returns `True` unless you've run out of result sets. Unlike `Read` though, you don't need to call it before you start processing the first result set, because that would be inconvenient for the most common usage. For example, maybe we wanted to get all of the movies someone directed and all of the roles that person has had with one command. Listing 12.8 shows how we could do this.

Listing 12.8 Getting multiple result sets from one command

```
>>> # 1 is David Fincher's id in the person table
>>> command = connection.CreateCommand()
>>> command.CommandText = (
'select id, title, released from movie where director_id = 1;'
'select m.title, r.character from role r, movie m '
'where person_id = 1 and r.movie_id = m.id')
>>> reader = command.ExecuteReader()
>>> while True:
...   while reader.Read():
...     print [reader[i] for i in range(reader.FieldCount)]
...   if not reader.NextResult():
...     break
[1, 'Fight Club', <System.DateTime [15/10/1999 00:00:00]>]
[2, 'Se7en', <System.DateTime [05/01/1996 00:00:00]>]
['Being John Malkovich', 'Christopher Bing']
```

Two different queries in the text of one command (annotates the `CommandText` lines)

Deal with all records in the current result set (annotates `while reader.Read():`)

Move to the next set of records, if any (annotates `if not reader.NextResult():`)

Results from the first query (annotates the first two result lines)

Results from the second query (annotates the last result line)

Notice that the result sets can have different structures. In listing 12.8, the first query results in rows with an integer, a string, and a date, while the second has rows with two strings.

Once you've finished with the results held by a `DataReader`, it's important to call the `Close` method.[6] This allows the database to release the result set; while the .NET runtime will eventually free a reader that's gone out of scope, that "eventually" might be a long time in the future, and you might run out of database connections in the meantime.

Suppose our database was being used behind a website running a competition between people in the movie industry, so we wanted to use it to keep running totals of the number of movies each person had directed and acted in. We have role_count and directed_count fields in the person table; when we insert a movie record we want to increment the directed_count for the director, and when we insert a role we want to increment the role_count for the actor.[7] We could do this in two steps, inserting the row and then updating the relevant total, but this means that at certain points the database totals will be wrong. Someone could get all the movies for Wes Anderson after we've inserted the record for *The Darjeeling Limited* but before we've updated his directed_ count. Worse, someone could trip over the power cord on our server between the insert and the update, and his total would stay wrong until someone noticed. We can avoid getting into an inconsistent state like this by using database transactions.

12.1.7 Using transactions

Transactions are a packaging mechanism that database engines provide to allow applications to make several changes to the tables in the database but have them appear to database users as if they all happened at once, at the moment the transaction is committed. *Committing* a transaction means applying the changes it contains. If something has gone wrong during a transaction (say an error in the program or some precondition for the changes we were making isn't met), we can throw away the changes in it by rolling back the transaction.

We can create a transaction using the `BeginTransaction` method of a connection.

```
>>> transaction = connection.BeginTransaction()
>>> transaction
<NpgsqlTransaction object at 0x…>
```

Until now, the connection we've been using has been in autocommit mode. This means that each command we've run has been wrapped in an implicit transaction that has been committed after the command has run. We can prevent this by explicitly associating a transaction with a command before executing it, as shown in listing 12.9.

---

[6] One way to ensure the reader is closed is the `with` statement. IronPython maps the Python context manager method `__exit__` to the `IDisposable` interface's `Dispose` method, and `DataReaders` and `Connections` both implement `IDisposable`.

[7] This denormalization is premature when the database is this size, but potentially in a large database where reads of the totals are much more common than writes to the movie and role tables, it would make sense.

Transaction properties: ACID

The key feature of transactions is that they give us four guarantees about how changes will be made to the database. (The guarantees are often referred to using the mnemonic ACID.) These guarantees are the following:

Atomicity—The changes made in a transaction are treated as all-or-nothing. If any of them are applied, they all are.

Consistency—A transaction cannot succeed if it would leave the database in an inconsistent state, for example, where a record in a child table has no corresponding parent record.

Isolation—Other transactions can't see changes made in this transaction until it has been committed.

Durability—Once a transaction has been committed, the changes it contains will be applied to the database, even in the event of system failure.

These properties make changes to databases much easier to reason about. In some circumstances maintaining the isolation constraint can be costly, so you can ask the database to reduce a transaction's isolation level for performance reasons.

Listing 12.9 Setting the transaction of a command before execution

```
>>> command = connection.CreateCommand()
>>> command.CommandText = ("insert into movie (title, released,"
... " director_id) values ('The Darjeeling Limited', '2007-11-23'"
... ", 2)")
>>> command.Transaction = transaction
>>> command.ExecuteNonQuery()
```

If you check in the database using psql or pgAdmin, the record for *The Darjeeling Limited* doesn't show up yet, because the transaction hasn't been committed. This allows us to update the person table, as shown in listing 12.10, to ensure the database state is consistent before any of it is visible.

Listing 12.10 Executing another command as part of the same transaction

```
>>> command = connection.CreateCommand()
>>> command.CommandText = ('update person set directed_count ='
... ' directed_count + 1 where id = 2')
>>> command.Transaction = transaction
>>> command.ExecuteNonQuery()
```

In listing 12.10 we create another command and execute it after associating it with the transaction we already have in progress.

Again, checking the database will show that the data is apparently unchanged. Now we can apply both changes at once by committing the transaction.

```
>>> transaction.Commit()
```

We could have canceled the changes by calling the transaction's `Rollback` method. If the application had crashed for some reason before we called `Commit`, the database would have discarded the changes. In fact, the point of the transaction machinery is to guarantee that even if the database server crashed, once it was brought back up, the database would still be in a consistent state.

Now that we've looked at transactions, we've covered the use of all of the key classes in the data provider layer bar one: the `DataAdapter`. Since the purpose of the `DataAdapter` is to connect the Data Provider layer with `DataSets`, to discuss them properly we need to look at `DataSets` as well.

12.1.8 DataAdapters and DataSets

The data provider classes provide functionality that will enable you to do anything you might need to with your database. However, for some uses they are inconvenient. `DataReaders` provide a read-only, forward-only stream of data; if you want to do complex processing that requires looking at data a number of times or navigating from parent records to child records, you'll need to perform multiple queries or store the information in some kind of data structure. While this data structure could be a collection of custom objects, for some applications it can be convenient to use a `DataSet`.

`DataSets` are DBMS independent, unlike data providers, so you don't need to use different classes for different databases. The details of how to communicate with a given database are specified by the `DataAdapter` class from the data provider.

So how do we get data into a `DataSet`? In listing 12.11, we ask the `DataAdapter` to fill it up.

Listing 12.11 Filling a `DataSet` from a `DataAdapter`

```
>>> clr.AddReference('System.Data')
>>> from System.Data import DataSet
>>> dataset = DataSet()
>>> adapter = pgsql.NpgsqlDataAdapter()
>>> command = adapter.SelectCommand = connection.CreateCommand()
>>> command.CommandText = 'select * from person'
>>> adapter.Fill(dataset)
21
```

Before the adapter can fill the `DataSet`, it needs to know how to get the data, which we tell it by setting its `SelectCommand`. The `Fill` method returns the number of records the command returned. In listing 12.12 we examine the information that came back from the database.

Listing 12.12 What's in the `DataSet` after filling it?

```
>>> dataset.Tables.Count
1
>>> table = dataset.Tables[0]
>>> table.TableName
'Table'
>>> table.Columns.Count
2
```

```
>>> [(c.ColumnName, c.DataType.Name) for c in table.Columns]
[('id', 'Int32'), ('name', 'String')]
>>> table.Rows.Count
7
>>> row = table.Rows[0]
>>> row['id'], row['name'] # you can access fields by name
(1, 'David Fincher')
>>> row[0], row[1] # or by index
(1, 'David Fincher')
```

As you can see, the `adapter.Fill` call has created a new table (rather unimaginatively named Table) in the `DataSet`. This gives us the basic structure of a `DataSet`: a collection of tables, each of which has columns defining the structure of the data and rows containing the data.

You can use the multiple-result-set capability of a command to fill more than one table in the dataset. You can see this behavior in listing 12.13.

Listing 12.13 Filling a `DataSet` with more than one result set

```
>>> dataset = DataSet()
>>> adapter = pgsql.NpgsqlDataAdapter()
>>> command = adapter.SelectCommand = connection.CreateCommand()
>>> command.CommandText = ('select * from person; '
... 'select * from movie; '
... 'select * from role')
>>> adapter.Fill(dataset)
21
>>> [t.TableName for t in dataset.Tables]
['Table', 'Table1', 'Table3']
>>> t = dataset.Tables
>>> t[0].TableName, t[1].TableName, t[2].TableName = 'person', 'movie', 'role'
>>> [(t.TableName, t.Rows.Count) for t in dataset.Tables]
[('person', 21), ('movie', 6), ('role'', 16)]
```

In listing 12.13, the `CommandText` we provide for the `SelectCommand` consists of three SQL statements to get data for all of the tables in the database at once. When we fill the `DataSet` from this, we get three tables, with the default names Table, Table1, and Table2. We then change the tables' names to match what they contain, and we can manipulate them in the same way we saw in listing 12.12.

`DataSet`s have a huge array of features beyond what we've seen here:

- Each `DataTable` can have `UniqueConstraints` and `ForeignKeyConstraints` to ensure that their data is manipulated only in consistent ways.
- A `DataSet` can be configured with `DataRelations`,[8] which specify parent-child relationships between tables (such as movie and role in our example database).
- The data in `DataSet`s can be updated and then sent back to the database using the `DataSet.Update` method. This uses the `InsertCommand`, `UpdateCommand`, and `DeleteCommand` properties on the `DataAdapter`.

[8] These should really be called `DataRelationships` to avoid confusion with relations (that is, tables) in the relational model sense.

`DataSets` are a large topic, and you can find a lot of information about them in books about ADO.NET and articles on the web. Despite that, in my experience working with databases and the .NET framework, the advanced features of `DataSets` tend not to be as useful as you might expect. They fall into a very narrow space between two broad types of tasks.

On one hand, tasks that require simple database interaction are better done using the Data Provider layer directly. This is even truer in IronPython than in languages like C# or VB.NET, because Python's rich native data structures make it easy to collect the results of a query as a list or dictionary of items (something you might do with a `DataTable` or `DataSet` otherwise).

On the other hand, for tasks needing complex database interaction, it's almost always better to use classes to model the entities involved. Then you can attach complicated behavior to the objects and give them methods to manage their relationships directly. In general, these classes will use the data provider classes to load and store their information in the database, while client code will deal with the higher-level interface the custom classes provide. The resulting code is clearer and easier to maintain because it's more closely related to the problem domain, rather than dealing with the verbose, very general API of the `DataSet`.

In section 12.1 we explored how you can use the different parts of ADO.NET with IronPython to interact with a wide range of data sources. There's a lot more to see, at both the `DataSet` and Data Provider layers, and lots of examples and articles are available that go into more detail on specific areas of the framework, as well as extensive MSDN documentation. Since ADO.NET is essentially a set of class libraries, examples using C# or VB.NET are easy to convert into IronPython code.

In the next section we look at how we can use IronPython to deal with another kind of data source: a web service. We cover talking to various kinds of web services from IronPython, as well as implementing one from the ground up.[9]

12.2 Web services

The term *web service* has a number of meanings, depending on whom you ask. The simplest is that a web service is a way for a program on one computer to use HTTP (and hence the web) to ask another computer for some information or to ask it to perform some action. The key idea is that the information that comes back in response is structured to be useful to a computer program, rather than the jumbled HTML of a normal web page with navigation, advertising, script code, and content all mixed together. The format of the information could be anything; commonly used formats are XML (particularly RSS, Atom, and SOAP), JavaScript Object Notation (JSON), and good-old plain text. Python's strong string-handling capabilities come in very handy when dealing with all of these different formats, and often libraries are available to do the parsing for us.

[9] Well, maybe not from the ground up. There are a lot of goodies to help in the .NET framework!

In section 12.2 we look at three different kinds of web service. To begin, we see an example of one of the simplest: the feed of recent articles from a weblog. The code for using this service makes exactly the same kind of request a web browser makes for web pages, but it processes the information it gets back rather than displaying it directly to the user. In section 12.2.2 we look at using SOAP web services, which have a different way of packaging up requests and responses. The .NET framework has some tools that make SOAP services very convenient to use. Then in the last part of the chapter we explore how we can build our own web service using the REST architectural style.

To use the first of our web services, we can request the Atom feed from a weblog to see the articles that have been posted recently.

12.2.1 Using a simple web service

When an article is published on a weblog, the blogging system will update its Atom or RSS feed so that news readers can alert users to the new article. We can download the feed using the `WebClient` class, which provides a high-level interface for talking to websites.

```
>>> from System.Net import WebClient
>>> url = 'http://www.voidspace.org.uk/ironpython/planet/atom.xml'
>>> content = WebClient().DownloadString(url)
```

Now content is a string containing XML in the Atom format. An Atom document looks something like what you see in listing 12.14.

Listing 12.14 Anatomy of an Atom XML weblog feed

```
<?xml version="1.0" encoding="utf-8" standalone="yes" ?>
<feed xmlns="http://www.w3.org/2005/Atom">

     <title>feed title</title>
     <link rel="self" href="link to feed"/>
     <link href="link to site"/>
     <id>unique identifier of the feed</id>
     <updated>when it was last updated</updated>

     <entry>
            <title type="html">title of this entry</title>
            <link href="url of this entry"/>
            <id>unique identifier of this entry</id>

            <updated>when it was last updated</updated>
            <content type="html">the body of this entry</content>

            <author>
                    <name>author's name</name>
                    <uri>author's website</uri>
            </author>
     </entry>

(more entries)

</feed>
```

The feed structure (in listing 12.14) has a small set of metadata elements, including these:

- Its title
- A link to the feed
- A link to the site that provides the feed
- A unique identifier for the feed
- When the feed was last updated

Then it has a number of entry elements, each of which has its own details:

- A title for the entry
- A link to the entry
- A unique identifier for this individual entry
- When it was updated
- The content of the entry
- The author's details: a name and a link to the author's website

We can process this in a number of ways; we could use libraries to parse Atom feeds and produce objects that give us the details of items in the feed, or we could use the `XmlDocumentReader` class from chapter 5 to extract the details. Those would work fine, but in web service scenarios we're often dealing with ad hoc XML formats, so it's handy to have a simple way of extracting details from the document without having to write specific handler classes for each format.

In that vein, in listing 12.15 we use a wrapper around `XmlElements` that makes pulling information out of arbitrary XML documents more convenient: it uses `__getattr__` to make failed attribute lookups try to find the name in XML attributes and child elements.[10] You can see the code for the `simplexml` module in the source code for section 12.2.1.

Listing 12.15 Extracting Atom feed details with `simplexml`

```
>>> from simplexml import SimpleXml
>>> feed = SimpleXml.fromString(content)          # Parses the feed XML
>>> feed.title                                    # Child elements are available as attributes
'Planet IronPython'
>>> feed.link.href                                # Navigate multiple elements at once
'http://www.voidspace.org.uk/ironpython/planet/atom.xml'
>>> len(feed.entries)                             # Elements appearing multiple times become a list
120
>>> for entry in feed.entries[:3]:
...   print entry.title.InnerText                 # Standard XmlElement attributes are also available
...   print entry.link.href
...   print entry.author.name
```

[10] This approach is based on the one used in the DynamicWebServices C# sample released by the IronPython team. That was released for IronPython 1.1, and the APIs it uses are not available in IronPython 2.0, but the behavior was fairly simple to reproduce in IronPython itself.

```
...   print
...
Miguel de Icaza: Custom Controls in Gtk# Article
http://tirania.org/blog/archive/2008/Jun-01-1.html
Miguel de Icaza (miguel@gnome.org)

Miguel de Icaza: Holly Gtk Widgets
http://tirania.org/blog/archive/2008/Jun-01.html
Miguel de Icaza (miguel@gnome.org)

John Lam on IronRuby: IronRuby and Rails
http://feeds.feedburner.com/~r/LessIsBetter/~3/301250579/ironruby-and-
➥   rails.html
John Lam
# and so on...
>>> fuzzy_entries = [e.title.InnerText for e in feed.entries
if e.author.name.startswith('Fuzzyman')]
>>> len(fuzzy_entries)
42
```

Selecting entries from the feed

You can see in listing 12.15 that getting details out of the feed XML is very easy using this technique (assuming you know the structure of the XML), and now you could store the entry data in a database or a file or display it in a user interface. Python's powerful list comprehensions can be very useful at this point for chopping up the data and filtering out desirable (or undesirable) parts.

A web service can be as straightforward as that: an occasionally updated file on a web server that is downloaded by programs running somewhere else. In general, though, a web service will wrap up a number of related operations on some data. In an example we'll look at soon, the operations will enable us to query, create, update, and delete a set of notes stored by the service.

There are two main approaches to web services: the SOAP camp and the proponents of a technique called REST. Although both of these techniques sit on top of HTTP, they use it in very different ways.

In a SOAP web service, all calls are HTTP POSTs to one URL. The posted data contains XML that identifies the operation that should be executed, as well as the parameters for the operation, in a structure called the *SOAP envelope* (because it has the address information for the message). The operations offered by a SOAP service and the types of the parameters they require are documented in an XML format called WSDL. WSDL service descriptions are fairly complicated structures, but the advantage of having them in a machine-readable format (rather than just as documentation) is that they can be created and used by tools. In general, this is the way SOAP services are made (particularly in .NET); the service provider will create an interface in code, which is then analyzed by a tool to produce WSDL for the service. Service consumers then take that WSDL and feed it to another tool, which generates a proxy to allow communication with the service.

Where a SOAP service uses only POSTs to a specific URL, with all other information carried in XML envelopes, REST services use a much broader range of HTTP features, and a service will handle an arbitrary number of URLs (generally the entire URL space

under a root). A URL within a REST service denotes a resource managed by the service. Each resource is manipulated by sending an HTTP request to the URL, and which operation to perform on the resource is determined by the HTTP method specified. Essentially an operation in a REST service is the combination of a noun (a resource URL) and a verb (the HTTP method).

> **HTTP methods**
>
> The HTTP standard defines a number of methods that can be specified in a request. These are the methods that are commonly used in REST services, along with what they might do:
>
> *`GET`*—Ask for the details of a resource.
>
> *`POST`*—Create, update, or delete a resource, or do *something* with the request data.
>
> *`PUT`*—Create or update a resource.
>
> *`DELETE`*—Delete a resource.
>
> There's obviously some overlap between `POST` and `PUT/DELETE`. In general the key difference is that `PUT` and `DELETE` act on specific individual resources, while a `POST` that created a new resource would act on the container. So, for example, if /users/john/bookmarks is the URL for John's bookmarks, you could create a new bookmark by `POST`ing a representation of a bookmark to a forum to that URL, but to update the bookmark you'd do a `PUT` to /users/john/bookmarks/forum. The meaning of `POST` is also more general, so it can be useful for things that don't fit so naturally into `GET/PUT/DELETE`, such as changing the workflow state of an item in a document management system.
>
> The standard also allows for defining new methods. Protocols such as WebDav do this, and it might be appropriate in some cases when defining the interface for a web service.

Those are the key differences between the approaches. Let's look at how we can use both kinds of web services from IronPython.

12.2.2 Using SOAP services from IronPython

The .NET framework has very solid tool support for SOAP/WSDL–style web services. In Visual Studio you can add a web reference to a web service URL that will automatically create a proxy class and all of the associated data structures that might be used in the service's interface. Under the hood, this uses a tool included with the .NET SDK called wsdl.exe to download the WSDL for the service and generate C# code, which is then compiled into an assembly that client code can use. We can use this method from the command line as well,[11] as you can see in listing 12.16:

[11] This uses a sample web service hosted by dotnetjunkies.com.

Listing 12.16 Generating a C# proxy for a SOAP service with wsdl.exe

```
C:\Temp> wsdl.exe /namespace:MathService http://www.dotnetjunkies.com/
    quickstart/aspplus/samples/services/MathService/
➥   CS/MathService.asmx
Microsoft (R) Web Services Description Language Utility
[Microsoft (R) .NET Framework, Version 2.0.50727.42]
Copyright (C) Microsoft Corporation. All rights reserved.
Writing file 'C:\Temp\MathService.cs'.
```

This creates `MathService.cs` and puts the generated class into the `MathService` namespace (which we need if we want to be able to import it easily). Now, as shown in listing 12.17, we can compile the code into a .NET assembly.

Listing 12.17 Compiling the C# proxy into a class library with csc.exe (the C# compiler)

```
C:\Temp> csc.exe /target:library MathService.cs
Microsoft (R) Visual C# 2005 Compiler version 8.00.50727.1433
for Microsoft (R) Windows (R) 2005 Framework version 2.0.50727
Copyright (C) Microsoft Corporation 2001-2005. All rights reserved.
```

We can use the generated proxy in the assembly to communicate with the web service as if it was a local library, as shown in listing 12.18.

Listing 12.18 Using the `MathService` proxy

```
>>> import clr
>>> clr.AddReference('MathService')
>>> from MathService import MathService
>>> service = MathService()
>>> service.Add(3, 4)
7.0
```

The proxy also provides asynchronous versions of the methods, as shown in listing 12.19.

Listing 12.19 Calling a web service method asynchronously

```
>>> def showResult(s, e):
...    print 'asynchronous result received:', e.Result

>>> service.MultiplyCompleted += showResult
>>> service.MultiplyAsync(4, 5)
>>> asynchronous result received: 20.0
```

Both of the command-line tools used here are actually using facilities for code generation and compilation that are built into the framework. All of the creation of proxy classes can be done at runtime, if necessary. In prototyping situations this can be quite handy. The code in listing 12.20 uses the `wsdlprovider` module, which you can find in the source code for section 12.2.2.

Listing 12.20 Calling a SOAP service using a dynamically generated proxy

```
>>> import wsdlprovider as wsdl
>>> url = 'http://www.webservicex.net/WeatherForecast.asmx'
```

```
>>> assembly = wsdl.GetWebservice(url)
>>> service = assembly.WeatherForecast()
>>> f = service.GetWeatherByZipCode('90210')
>>> print f.PlaceName, f.StateCode
BEVERLY HILLS CA
>>> for d in f.Details:
...   print d.Day, d.MaxTemperatureC
...
Sunday, June 08, 2008 24
Monday, June 09, 2008 24
Tuesday, June 10, 2008 23
Wednesday, June 11, 2008 24
Thursday, June 12, 2008 27
Friday, June 13, 2008 27
```

That gives you an overview of how to use SOAP web services from IronPython. How about implementing them? Unfortunately, that's very difficult. The .NET framework's support for creating SOAP services is primarily based on introspecting classes and using attribute annotations on them to generate the web service description and hook your custom classes into the server. Since IronPython doesn't create classes that are directly usable from the C# side, the tools can't process them, and there's no way for us to attach attributes to them anyway. In theory, we could use the XML libraries to roll our own WSDL and SOAP interfaces, but given the complexity of the formats, this would be a lot of work.

This doesn't mean we can't implement web services in IronPython at all, though. REST services are based on the idea of reusing the features of HTTP, rather than simply sitting on top of it in the way that SOAP does. This means that a lot of the basic structure for implementing a REST service is already provided by the components that handle normal web traffic in the .NET framework. Let's take a look at how this works.

12.2.3 REST services in IronPython

Rather than looking at how to use a public web service (which will probably only have read-only access), we're going to implement our own basic but fairly complete REST service in IronPython. This service will store notes that are given to it by client programs. Notes are simple data items containing the following:

- A title
- A body
- An id—a unique identifier for the note that allows it to be retrieved

The next step in defining our service is to determine the operations we want to support and how we will map the operations to resource URLs and methods. Table 12.2 provides a description of each operation in the notes service.

As you can see, some requests pass a note as a payload. After each request, the server indicates whether the operation was completed using the HTTP status code and sends back a response that contains the information that needs to be sent back (if any). In addition, each response will have a status (normally `ok`, if the operation succeeded), and in the case that the operation failed, a reason will be given.

Table 12.2 The operations provided by the notes service

Operation	URL	Method	Details
List notes	/notes	GET	Response will contain a list of note titles and links.
Add a note	/notes	POST	Request should be the note to add. Response will contain a link to the new note.
Get the details of a note	/notes/<note id>	GET	Response will contain the details of the note.
Update a note	/notes/<note id>	PUT	Request should be the updated note.
Delete a note	/notes/<note id>	DELETE	

This set of operations and rules almost gives us an interface that we could write a client for. The only detail that remains is what the formats of the various payloads should be. This could actually be any format we like; XML is an obvious one, but JavaScript Object Notation (see http://json.org) or plain text could work just as well. We stick with XML since in .NET it doesn't require any extra libraries.

So the payloads we have in requests and responses are notes, links to notes, and responses.

- Notes will look like this:

```
<note id="note_id" title="the title of the note">
Note body
</note>
```

- Links to notes will look like this:

```
<notelink title="note title" href="http://url/of/note"/>
```

- Responses will look like this:

```
<response status="ok or error">
Either a note, note link, list of links or error reason, depending on the
request.
</response>
```

The response element will always be the root of the XML document the service sends back. It acts as a container for the information we need to send back, and it provides a handy place to report status information that isn't part of the notes that we're managing.

With the formats nailed down, we have a complete interface. The service and clients could be implemented in any language, as long as it has facilities for talking HTTP and parsing and generating XML, although obviously we'll be looking at implementations in IronPython. Let's look at how a client for this service would be written.

THE NOTES SERVICE CLIENT MODULE

The client module abstracts the communication and message parsing that we need to do to make requests to the notes service, so that we can write applications that use the notes service without worrying much about the underlying details. Essentially, it's a hand-written version of the code the .NET WSDL tools generate for a SOAP service, but because the REST style is so close to HTTP, the code is quite simple.

We start from the core of the module in listing 12.21 and work outward. The complete `client.py` module is available in the source code for this section. It uses the `Note` class from `note.py`, which stores the note's id, title, and body, and provides methods to convert to and from XML. The notes service client also uses the `simplexml` module we saw earlier in the chapter for parsing responses.

Listing 12.21 The notes service client: `sendMessage`

```
from simplexml import SimpleXml
from note import Note

from System.IO import MemoryStream
from System.Net import WebException, WebExceptionStatus,\
   WebRequest
from System.Xml import XmlDocument, XmlWriter        # For creating and writing payloads

def sendMessage(method, url, note=None):
   request = WebRequest.Create(url)
   request.Method = method
   if note is not None:                              # Not all operations need notes
      writeNoteToRequest(note, request)
   try:
      response = request.GetResponse()               # Sends request data to the server
   except WebException, e:
      if e.Status == WebExceptionStatus.ProtocolError:   # Indicates an HTTP error code
         # this is an error message from the service
         # look in the response for the problem
         response = e.Response
      else:
         raise
   stream = response.GetResponseStream()
   responseDoc = SimpleXml.fromStream(stream)        # Parses the response into a SimpleXml
   stream.Close()
   checkResponse(responseDoc)
   return responseDoc

class ServiceError(Exception):
   pass

def checkResponse(response):
   if response.status != 'ok':                       # Checks the status attribute of the response
      raise ServiceError(response.InnerText)
```

The `sendMessage` function in listing 12.21 is the heart of the client module. It takes an HTTP method (like `GET` or `PUT`), the URL to send the request to, and a `Note` instance, if one should be sent as the payload of this message. It sends the request, parses the response, checks to see whether the operation was successful (using the `checkResponse` function), and returns the response. The higher-level functions on top of `sendMessage` will then be able to get the details they need out of the response node.

Next, in listing 12.22 we have the `writeNoteToRequest` function, which handles the nuts and bolts of creating an XML document and serializing it into the request:

Listing 12.22 Sending a note to the server from the client

```
def writeNoteToRequest(note, request):
   doc = XmlDocument()
   doc.AppendChild(doc.CreateXmlDeclaration('1.0', 'utf-8', 'yes'))
   doc.AppendChild(note.toXml(doc))
   request.ContentType = 'text/xml'
   stream = request.GetRequestStream()
   writer = XmlWriter.Create(stream)
   doc.WriteTo(writer)
   writer.Flush()
   stream.Close()
```

We add a declaration to the document to indicate the encoding, and then we convert the note to XML nodes and add it to the document. Notice that we pass the document into the call to `note.toXml`; this is because the `XmlDocument` has methods for creating XML nodes that are associated with the document. There's no simple way of creating the nodes without the document they are going to be added to.

With `sendMessage` as a base, the implementations of functions to call the different note service operations follow very simply from the definition of the service interface, as you can see in listing 12.23.

Listing 12.23 The notes service client: note operations

```
index = 'http://localhost:8080/notes'

def getAllNotes():
   response = sendMessage('GET', index)
   return [(n.title, n.href) for n in response.notelinks]

def getNote(link):
   response = sendMessage('GET', link)
   return Note.fromXml(response.note.element)

def addNote(note):
   response = sendMessage('POST', index, note)
   return response.notelink.href

def updateNote(link, note):
   sendMessage('PUT', link, note)

def deleteNote(link):
   sendMessage('DELETE', link)
```

Each function in listing 12.23 calls `sendMessage` with the method and URL that represent its operation, as well as a note if one should be sent with the request, and extracts the information it needs from the response that comes back.

At this point the client is complete; it provides a function for every operation the notes service offers. If you look in `client.py`, you can see some examples of using the module to manipulate notes in the service. Of course, you won't be able to run them without the service itself! So now it's time to see how it hangs together.

THE NOTES SERVICE

The notes service is built on the .NET `HttpListener` class, which provides a simple way to write applications that need to act as HTTP servers. To use the `HttpListener`, you

tell it what URLs you want to handle, start it, and wait for requests. Each request comes as a context object that gives you access to everything that was sent from the client and allows you to send response data back. The `HttpListener` provides methods for handling requests asynchronously, but to keep things simple the notes service uses the synchronous `GetContext` method to receive requests. The full code of the `NotesService` class can be found in `service.py` in the source code for section 12.2.3. First, in listing 12.24 we look at the main loop of the service.

Listing 12.24 The notes service main loop

```
class NotesService(object):

   def __init__(self):
      self.listener = HttpListener()
      self.listener.Prefixes.Add('http://localhost:8080/notes/')
      self.notes = {
         'example': Note('example',
                         'An example note',
                         'Note content here')}

   def run(self):
      self.listener.Start()
      print 'Notes service started'
      while True:
         context = self.listener.GetContext()
         self.handle(context)
```

As you can see, when a `NotesService` instance is created, we create its `HttpListener` and specify that we want to handle all requests that begin with `'http://localhost:8080/notes/'`. We also create a dictionary that acts as storage for all notes maintained through the service. Obviously this could be replaced by storing the notes in a database or in the file system in a more full-featured implementation. In the `run` method, we have the main loop of the service. The `HttpListener` is told to begin listening for incoming requests, and we call its `GetContext` method in an infinite loop. `GetContext` will block until a request is received. Then we call the `handle` method, which uses the HTTP method of the request and the requested URL to dispatch the request to the service's operation methods. Let's look at the `handle` method now in listing 12.25.

Listing 12.25 `NotesService`: dispatching requests in the `handle` method

```
def handle(self, context):
   path = self.pathComponents(context)
   method = context.Request.HttpMethod
   print method, path
   handlers = {}
   if path == ['notes']:                                   <- Possible handlers for requests to /notes/
      handlers = dict(GET=self.getNotes, POST=self.addNote)
   elif len(path) == 2:                                    <- Possible handlers for /notes/note-id
      handlers = dict(GET=self.getNote,
                      PUT=self.updateNote,
```

```
                    DELETE=self.deleteNote)
    handler = handlers.get(method, self.error)     <-- Fall back to the error handler if none match
    try:
        handler(context)     <-- Call the selected handler with context
    except ServiceError, e:
        self.error(context, e.statusCode, e.reason)     <-- Send the error back to the client
```

First, `handle` gets the requested path (as a list of the slash-separated components in the URL) and the HTTP verb that was used in the request. For each kind of path we build a dictionary of handlers, and then we use that handler dictionary to get the method corresponding to the verb that was sent. The handler dictionary maps the verbs that are valid for that kind of path to their corresponding `NotesService` methods. In the event that we were given a totally invalid path or a verb that isn't supported, we dispatch the request to the `error` method, which sends a response indicating that the combination of method and path didn't make sense. If there is a problem while one of the handlers is executing (for example, if a client tries to update a note that doesn't exist), the handler will raise a `ServiceError` exception (defined elsewhere in `service.py`) with the HTTP status code that should be sent, as well as a message describing the problem.

The service can now dispatch requests to the correct handler method based on the verb and URL. What does a handler method look like? They all have the same basic structure:

1. Gather any information needed from the request and check that the operation makes sense (no deleting nonexistent notes!).
2. Perform the requested operation (if it's something that changes the stored notes).
3. Create a response document with any information that needs to be sent back to the client and send it.

Making a response and writing it back to the client are common tasks, so they have been pulled out into separate methods. `makeResponse` creates an XML document containing an empty response element with a default status of `ok`. `writeDocument` handles the nuts and bolts of writing the XML document out to a response stream. They're quite similar to the code for building documents and writing them to streams that we saw in the client module, so we won't cover them here. You can see the details in the source code for the `NotesService` class.

The first handler we will look at is `getNotes`, which generates a response with links to all of the notes. You can see what it looks like in listing 12.26.

Listing 12.26 `NotesService`: the `getNotes` handler

```
def getNotes(self, context):
    doc = self.makeResponse()
    for note in self.notes.values():
        link = note.toSummaryXml(
            doc, LINK_TEMPLATE)
```

```
      doc.DocumentElement.AppendChild(link)
   self.writeDocument(context, doc)
```

Since this operation is only a query, we don't need to change the notes store. We create a new response document and then go through all of the notes, creating an XML snippet representing a link to each (using the `LINK_TEMPLATE` constant defined at the top of the file) and adding that to the response, before sending it back using `writeDocument`.

The other handler that deals with requests to the top-level URL, /notes/, is `addNote`, in listing 12.27. The method needs a note to be sent in the request, so we need the helper method `getNoteFromRequest` to parse the XML document into a note.

Listing 12.27 `NotesService`: the `addNote` handler

```
def getNoteFromRequest(self, request):
   message = XmlDocument()
   message.Load(request.InputStream)
   request.InputStream.Close()
   return Note.fromXml(message.DocumentElement)

def addNote(self, context):
   note = self.getNoteFromRequest(context.Request)
   if note.id in self.notes:
      raise ServiceError(400, 'note id already used')
   self.notes[note.id] = note
   doc = self.makeResponse()
   doc.DocumentElement.AppendChild(note.toSummaryXml(doc, LINK_TEMPLATE))
   self.writeDocument(context, doc)
```

Once `addNote` has read in the note that was sent, it checks that the id is not already taken. If it is, it raises a `ServiceError`, which is caught in `handle` and causes an error to be written back to the client with the problem. Otherwise, we add the note to the store and send the client back a link to the new location of the note.

The next operation to look at is `getNote`, which is the first that handles a URL for an individual note. We have three operations at this level, so we obviously also need a method to get the note that is referred to by the current URL from the store, which we'll call `getNoteForCurrentPath`. You can see these methods in listing 12.28.

Listing 12.28 `NotesService`: the `getNote` handler

```
def getNoteForCurrentPath(self, context):
   lastChunk = self.pathComponents(context)[-1]
   noteId = HttpUtility.UrlDecode(lastChunk)
   note = self.notes.get(noteId)
   if note is None:
      raise ServiceError(404, 'no such note')
   return note

def getNote(self, context):
   note = self.getNoteForCurrentPath(context)
   doc = self.makeResponse()
   doc.DocumentElement.AppendChild(note.toXml(doc))
   self.writeDocument(context, doc)
```

`getNoteForCurrentPath` gets the path and uses the `HttpUtility.UrlDecode` method to decode the note id in the URL. This is because some characters have special meaning in URLs and so can't be used "naked" in ordinary parts of a URL, like the note id in a request to the notes service. These special characters are quoted by replacing them with a `%` and then their ASCII code in hexadecimal. So forward slashes become `%2F` (because `ord(/)` is 47, which is `0x2F`), and spaces become `%20` (although sometimes they are special-cased as `+` for readability). Once we have the decoded note id, we check to see whether there is a note with that id in the store; if there isn't, we raise an HTTP 404 Not Found error. Once the note is retrieved, `getNote` is simple: it just writes the note out to the client.

The next operation is `updateNote`, in listing 12.29, which brings together all of the building blocks we've seen: it receives both a note in the request and a note id in the URL.

Listing 12.29 `NotesService`: the `updateNote` handler

```
def updateNote(self, context):
   note = self.getNoteForCurrentPath(context)
   updatedNote = self.getNoteFromRequest(context.Request)
   if note.id != updatedNote.id:
      raise ServiceError(400, 'can't change note id')
   self.notes[note.id] = updatedNote
   self.writeDocument(context, self.makeResponse())
```

`updateNote` gets the note for the current path from the storage and then reads in the note that has been sent. We check that the id has been kept the same; if not, we raise an error. This is a slightly arbitrary restriction, although there is some justification for it: one of the principles of the web (which is inherited by REST) is that things—resources—shouldn't move. The key feature of the web is linking between resources, and some of those links may be from places that you can't update. Allowing the note id to change would change its URL, so any links to the old note would be broken.

If the note exists in the store and the new one has the same id, we replace the old note in the store with the new one. The update operation doesn't need to return anything (other than telling the client that everything was OK), so it just writes an empty response back.

The last operation is `deleteNote` (listing 12.30), which is simple.

Listing 12.30 `NotesService`: the `deleteNote` handler

```
def deleteNote(self, context):
   note = self.getNoteForCurrentPath(context)
   del self.notes[note.id]
   self.writeDocument(context, self.makeResponse())
```

We get the note, and if it exists, we remove it from the `NotesService` store. Then we tell the client that the job's been done.

And that's the last operation we wanted to support! Now the notes service is complete. You can see the full code for the `NotesService` class in the source code for section 12.2.3. If you want to see it in action, you can run it at the command line with `ipy service.py`; when it's started and listening for requests, you'll see the message `Notes service started`. The `client` module has some examples of using the service in its `if __name__ == '__main__'` section, so you can run it with `ipy client.py`. When you run the client, the server will show you the requests it's receiving as they come in.

Another trick that can be quite useful when trying out libraries like the `client` module is the `-i` command-line option for ipy.exe,[12] making the command `ipy -i client.py`. (The *i* stands for *interact*.) This will run the file, including the `__main__` section, and then display a normal Python `>>>` prompt instead of exiting, which lets you run commands in the module. So you can try calling `getAllNotes` and then creating your own `Notes`, sending them to the service with `addNote`, and manipulating them with the other functions in the module.

Now you've seen how we can communicate with and implement REST-style web services using IronPython. Obviously our notes service is fairly simple; it doesn't store the notes anywhere so they won't persist between runs of the service. It can handle only a single request at a time, and it doesn't have the kind of error handling or logging you would want in a production system. That said, it covers a lot of the techniques you'll need to create your own services, and you can apply the principles of the REST architectural style it uses to many situations where you need separate systems to communicate over the web.

There are philosophical differences between the SOAP and REST techniques for building web services. With SOAP, the focus is on using tools to automatically generate web services and clients from interfaces or class descriptions, which can be very convenient. However, frequently the tools for different platforms or vendors (such as Microsoft, Sun, and IBM) will not agree on how an interface or data structure should be represented in WSDL or SOAP, and so services created with the tools won't interoperate. If you're trying to integrate a Java system with a .NET one (and this kind of scenario is often exactly where you would want to use web services), the incompatibilities that arise can be extremely difficult to work around. And if you're trying to use a SOAP web service from a platform that doesn't have much tool support for it, the complexity of the protocol means that you have a *lot* of work to do. The REST style is in part a response to these problems: it's much simpler, so it's feasible to write the interfaces yourself, and you can easily debug and fix problems.

At the moment, neither style is obviously better in all situations. Luckily, we can use both kinds of services from IronPython. We can also create REST services, and in the case of creating SOAP services, a lot of effort is being devoted to the problem of adding .NET attributes to IronPython classes, both by the IronPython team and the com-

[12] This option works the same way in CPython as well.

munity.[13] Hopefully this work will enable us to use the .NET tools to build SOAP services as conveniently as we can use them.

12.3 Summary

In this chapter, we've covered the basics of using the ADO.NET libraries with IronPython to get access to data stored in all kinds of relational databases, from embedded databases like Sqlite all the way to high-end commercial databases like SQL Server and Oracle. The nice thing about the Data Provider layer is that it lets us talk to these very different databases with the same interface.

We've also looked at dealing with SOAP and REST web services from IronPython, as well as implementing our own simple REST service. Web services are becoming more and more useful in all kinds of ways. Companies such as Google, Amazon, and Yahoo! are exposing huge swathes of their systems as web services, so the information they provide can be mashed up and integrated more tightly with other websites, as well as enabling us to use their data in new ways. At the other end of the scale, we can build web services that provide back-end data to the browser, letting us make web applications with richer interfaces using AJAX and Flash.

In the next chapter, we look at Silverlight, a new browser plugin that can be used to make more dynamic web interfaces in a similar way to Flash. Of course, from our perspective, the key feature of Silverlight is that it allows us to script the browser in IronPython!

---

[13] The Coils project (http://www.codeplex.com/coils) is one approach to solving this problem.

13 Silverlight: IronPython in the browser

This chapter covers

- Packaging a dynamic Silverlight application
- Using the Silverlight APIs
- Building user interfaces
- Interacting with the browser DOM

Improved browser capabilities, enabled by faster computers, have not only made complex and rich web applications possible, but have also changed the way we use computers. Techniques like AJAX, and powerful JavaScript libraries that abstract away painful cross-browser issues, have resulted in client-side (in the browser) programming becoming one of the most important fields in modern software development. Despite this, web applications remain restricted by the user interface capabilities of JavaScript and the performance of JavaScript when running large programs. Ways around these difficulties include web-programming frameworks such as Flash, AIR, Flex, and Silverlight, which have their own programming and user interface models.

In case you haven't heard of it, Silverlight is a cross-platform browser plugin created by Microsoft. It is superficially similar to Flash but with the magic extra ingredient that we can program it with Python. Because Silverlight is based on the .NET framework, it is a major new platform that IronPython runs on.

Silverlight has a user interface model based on Windows Presentation Foundation, and we can use it to do some exciting things, such as media streaming, creating games, and building rich internet applications.

In this chapter, we look at some of what Silverlight can do and how to do it from IronPython. We'll be exploring, creating, and deploying dynamic Silverlight applications and programming with the Silverlight APIs—some of which are familiar and some of which are new.

13.1 Introduction to Silverlight

Silverlight is a cross-platform, cross-browser plugin for embedding into web pages. There are two versions of Silverlight: Silverlight 1 is for media streaming and is programmed with JavaScript. Silverlight 2, the interesting version, has at its heart a cut-down and security-sandboxed version of the .NET framework called the CoreCLR. That means it contains many of the APIs we are already familiar with, including the WPF user interface. More important, the CoreCLR is capable of hosting the Dynamic Language Runtime, so Silverlight can be programmed with DLR languages like IronPython and IronRuby.

In this chapter we explore some of the features that Silverlight provides. Figure 13.1 shows a Tetrislite[1] game written for Silverlight 2, from the Silverlight Gallery.

By *cross-platform*, Microsoft means Windows and Mac OS X. By *cross browser*, the Microsoft folks mean the Safari, Firefox, and Internet Explorer web browsers. This isn't the end of the story, though; Silverlight support is in the works for the Opera



Figure 13.1 Tetrislite from the Silverlight Gallery

[1] See http://silverlight.net/Community/gallerydetail.aspx?cat=sl2&sort=1.

browser and for the Windows Mobile and Nokia S60 platforms. The Silverlight team members are not working directly to support Linux, but they are working with the Mono team on an officially blessed Mono port of Silverlight called Moonlight.[2] This will initially work on Firefox on Linux, but the eventual goal is to get Moonlight working on multiple browsers (like Konqueror) and on every platform that Mono runs on.

Microsoft is assisting the Moonlight effort by providing access to the Silverlight test suite and the proprietary video codecs that Silverlight uses. At the time of writing, Moonlight supports the 1.0 engine, and work has begun on the 2.0 engine. Moonlight uses the Mono stack, but a lot of the work involves implementing the security model that provides the Silverlight browser sandboxing. Another big part is the user interface model; this is particularly interesting, as previously the Mono team has said that they have no interest in implementing WPF. Perhaps this will change now that they are implementing a subset of WPF for Moonlight.

So what benefits for web application programming does Silverlight have over traditional JavaScript and AJAX? The first advantage is that it can be programmed in Python, and frankly that's enough for us. The Python community has long wanted to be able to script the browser with Python rather than JavaScript, and it is at least slightly ironic that it is Microsoft that has made this possible. Perhaps a more compelling reason is that Silverlight is fast. By some benchmarks[3] IronPython code running in Silverlight runs two orders of magnitude faster than JavaScript! As well as having its own user interface model, Silverlight also gives you full access to the browser DOM (Document Object Model), so that everything you can do from JavaScript you can also do from inside Silverlight. In fact, one of the dynamic languages that run on the DLR is an ECMA 3[4]–compliant version of JavaScript called Managed JScript. This makes it easier to port AJAX applications to Silverlight, because a lot of the code can run unmodified.

The features of Silverlight include

- A user interface model based on WPF
- APIs including threading, sockets, XML, and JSON
- A powerful video player
- Deep zoom and adaptive media streaming
- Client-side local storage in the browser
- Access to the browser DOM and techniques to communicate between JavaScript and the Silverlight plugin

The last point is particularly interesting. Although most example Silverlight applications take over the whole web page, this is not the only way to use it. Like Flash, the Silverlight plugin can occupy as little of a web page as you want,[5] and you can embed several plugins (which can communicate with each other) in the same page. In fact, the Silverlight plugin need not be visible at all. By interacting with JavaScript and the

[2] See http://www.mono-project.com/Moonlight.

[3] And of course all benchmarks are misleading, in the same way that all generalizations are wrong.

[4] ECMA 3 is the standard that covers JavaScript 2, the version in use by most current browsers.

[5] Yes, you can use Silverlight to create really annoying adverts that everybody hates.

> **Deep Zoom and adaptive streaming**
>
> Deep Zoom is a fantastic image-zooming technology based on Seadragon: see http://livelabs.com/seadragon/.
>
> Adaptive streaming allows the browser to adjust the quality of streamed audio and video according to the available bandwidth and CPU power.

browser DOM, you can use all your favorite AJAX tricks from Silverlight, including using existing JavaScript libraries for the user interface, with the business logic implemented in Python.

Moonlight has an additional mode of operation that isn't (yet!) supported by Silverlight. Moonlight applications called desklets[6] can run outside the browser with full access to the Mono stack.

Creating Silverlight applications with dynamic languages is very simple. Let's dive into the development process.

13.1.1 Dynamic Silverlight

Silverlight applications are packaged as xap files (compressed zip files), containing the assemblies and resources used by your application. An IronPython application is just a normal application as far as Silverlight is concerned; a DLR assembly provides the entry point and is responsible for running the Python code.

The DLR assemblies, along with the tool for developing and packaging dynamic applications, are known by the wonderful mouthful *Silverlight Dynamic Languages SDK* and can be downloaded from http://www.codeplex.com/sdlsdk.

> **Chiron and the dynamic experience**
>
> Chiron packages IronPython Silverlight applications as the browser requests them. This means that you can have a truly dynamic experience developing Silverlight applications with IronPython. Simply edit the Python file with a text editor or IDE and refresh the browser, and you immediately see the changes in front of you.

The Silverlight Dynamic Languages SDK is comprised of the DLR runtime and IronPython, IronRuby, and Managed JScript assemblies compiled for Silverlight, along with the development tool Chiron.exe. Releases of IronPython should also include binary builds of IronPython for Silverlight and Chiron but won't include the other languages.

Chiron can create xap files for dynamic applications and will also work as a server, allowing you to use your applications from the filesystem while you are developing them. Chiron runs under Mono, so you can use it on the Mac.

Silverlight lives on the web, so to use it we need to embed it into a web page.

---

[6] See http://tirania.org/blog/archive/2007/Jun-28.html, but also see the following page for an alternative that works with Silverlight as well as Moonlight: http://blogs.microsoft.co.il/blogs/tamir/archive/2008/05/02/stand-alone-multiplatform-silverlight-application.aspx.

EMBEDDING THE SILVERLIGHT CONTROL

Embedding a Silverlight control into a web page is straightforward. You use an HTML `<object>` tag, with parameters that initialize the control and HTML that displays the Install Microsoft Silverlight link and image (shown in figure 13.2) if Silverlight is not installed.



Figure 13.2 The image displayed to the user if Silverlight is not installed

Listing 13.1 is the typical HTML for embedding the Silverlight control into a web page.

Listing 13.1 The HTML to embed the Silverlight control

```
<div id="SilverlightControlHost" width="480" height="320">

  <object data="data:application/x-silverlight,"
    type="application/x-silverlight-2"
    id="SilverlightControl"
    width="100%" height="100%">
    <param name="source" value="app.xap"/>
    <param name="onError" value="onError" />
    <param name="background" value="white" />
    <param name="initParams"
     value="debug=true,reportErrors=errorLocation" />

    <a href="http://go.microsoft.com/fwlink/?LinkID=124807"
      style="text-decoration: none;">
       <img src="http://go.microsoft.com/fwlink/?LinkId=108181"
       alt="Get Microsoft Silverlight"
       style="border-style: none" />
    </a>
  </object>
  <iframe style='visibility:hidden;height:0;width:0;border:0px'
></iframe>

</div>
```

- Required version of Silverlight
- Object tag that embeds the plugin
- Name of the application to load
- JavaScript function called in event of error
- Special handling for reporting errors
- Image/link shown if Silverlight is not installed

The `onError` parameter is a JavaScript function that will be called if an error occurs inside Silverlight (including in your code). The value in `initParams` is a bit special. `debug=true` enables better exception messages. As well as getting the error message, you'll get a stack trace and a snapshot of the code that caused the exception. It works in conjunction with an `errorLocation` div in your HTML (plus a bit of CSS for styling), which is where the tracebacks from your application are displayed. You can see the necessary HTML/CSS in the sources of the examples for this chapter.

CLR tracebacks in error reporting

By adding `exceptionDetail` to `initParams`, you can extend the error tracebacks to include the CLR errors:

```
<param name="initParams" value="debug=true, reportErrors=errorDiv,
exceptionDetail=true" />
```

This is *usually* more information than you want in tracebacks, but it can be invaluable in tracking down the underlying reason for some errors.

THE XAP FILE

The xap file is really just a zip file with a different extension; you can construct xap files manually or they can be autogenerated for you by Chiron. The command-line magic to make Chiron create a xap file from a directory is

```
bin\Chiron /d:dir_name /z:app_name.xap
```

If you're running this on the Mac, then the command line will look something like this:

```
mono bin/Chiron.exe /d:dir_name /z:app_name.xap
```

More important, you can use Chiron to test your application from the filesystem, without having to create a xap file. If your application is called app.xap, then your application files should be kept in a directory named app. Chiron will act as a local server and dynamically create xap files as they are requested. The command to launch Chiron as a web server is

```
Chiron /w
```

By default, this serves on localhost port 2060. The most important part of the xap file is the entry point, which in dynamic applications will be a file called app.py, app.rb, or app.js, depending on which dynamic language you are programming in.

13.1.2 Your Python application

Our first app.py will be the simplest possible IronPython Silverlight application. Because we're using the WPF UI system, we work with classes from `ystem.Windows` namespaces. Listing 13.2 creates a `Canvas`, with an embedded `TextBlock` displaying a message.

Listing 13.2 A Simple IronPython application for Silverlight

```
from System.Windows import Application
from System.Windows.Controls import Canvas, TextBlock

canvas = Canvas()
textblock = TextBlock()
textblock.FontSize = 24
textblock.Text = 'Hello World from IronPython'
canvas.Children.Add(textblock)

Application.Current.RootVisual = canvas
```

The important line, which is different from our previous work with WPF, is the last one, where we set the container canvas as the `RootVisual` on the current application. This makes our canvas the root (top-level) object in the object tree of the displayed user interface. You're not stuck with a single Python file for your application though. Imports work normally for Python modules and packages contained in your xap file (or your application directory when developing with Chiron). You can use `open` or `file` to access other resources from inside the xap file, but they are sandboxed, and you can't use them to get at anything on the client computer.

Project structure in Silverlight applications

In Silverlight projects you can break your applications into multiple Python files in the *same* way you can with normal Python applications. Import statements from Python files kept in the xap file work normally, including from Python packages. You can even keep Python files in a subdirectory and add it to `sys.path` to be able to import from them.

Because we're using WPF, many applications load XAML for the basic layout. They do so with very similar code to listing 13.2, as shown in listing 13.3.

Listing 13.3 IronPython Silverlight application that loads XAML

```
from System.Windows import Application
from System.Windows.Controls import Canvas

canv = Canvas()
xaml = Application.Current.LoadRootVisual(canv, "app.xaml")
xaml.textblock.Text = 'Hello World from IronPython'
```

Instead of setting the `RootVisual` on the application, this code loads the XAML with a call to `LoadRootVisual`. Listing 13.4 shows the app.xaml file, for a `Canvas` containing a `TextBlock`, loaded from the xap file in listing 13.3.

Listing 13.4 Silverlight XAML for UI layout

```
<Canvas x:Class="System.Windows.Controls.Canvas"
   xmlns="http://schemas.microsoft.com/client/2007"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">

<TextBlock x:Name="textblock" FontSize="30">Hello world from XAML</TextBlock>

</Canvas>
```

The call to `LoadRootVisual` returns us an object tree. Like the object trees we worked with from XAML in chapter 9, we can access elements that we marked with `x:Name` as attributes on the object tree. This allows us to set the text on the `TextBlock` through `xaml.textblock.Text`. Many of the techniques we learned when working with the WPF libraries for desktop applications are relevant to Silverlight, but they are far from identical. One of the major differences is that we have fewer controls to work with. Let's look at some of the APIs available for creating Silverlight applications, including the extended set of controls.

13.1.3 Silverlight controls

Silverlight ships with a set of standard controls based on WPF and contained in the `System.Windows.Controls` namespace. Figure 13.3 shows examples of the standard controls.

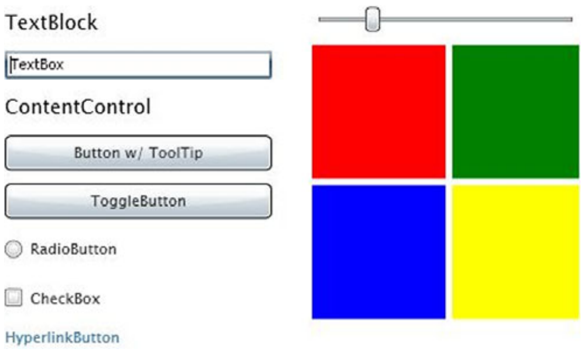


Figure 13.3 Some of the standard Silverlight controls

USING CONTROLS

As you might expect from their WPF inheritance, the Silverlight controls are attractive and easy to use. Figure 13.4 shows a `TextBox` with a `Button` and a `TextBlock`.

Using and configuring the controls from code is splendidly simple. Listing 13.5 shows a `Button` and a `TextBox` inside a horizontally oriented `StackPanel`. When the `Button` is clicked, or you press Enter, a message is set on the `TextBlock`.

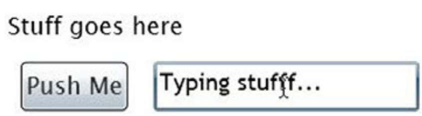


Figure 13.4 A `TextBox` with `Button` and `TextBlock`

Listing 13.5 A `TextBox` with a `Button` in a horizontal `StackPanel`

```
from System.Windows import Thickness
from System.Windows.Controls import (
  Button, Orientation, StackPanel, TextBox
)
from System.Windows.Input import Key

panel = StackPanel()
panel.Margin = Thickness(20)
panel.Orientation = Orientation.Horizontal

button = Button()
button.Content = 'Push Me'
button.FontSize = 18
button.Margin = Thickness(10)

textbox = TextBox()
textbox.FontSize = 18
textbox.Margin = Thickness(10)
textbox.Width = 200

def onClick(s, e):
  textblock.Text = textbox.Text
  textbox.Text = ""

def onKeyDown(sender, e):
   if e.Key == Key.Enter:
      e.Handled = True
```

Setting a margin around the element

Make the panel horizontal

Create the TextBox

```
    onClick(None, None)

button.Click += onClick
textbox.KeyDown += onKeyDown
```

Along with the standard controls, there is a set of extended controls that comes with Visual Studio Tools for Silverlight.[7] This set includes additional controls such as `Calendar`, `DataGrid`, `DatePicker`, `TabControl`, and so on.

TIP In addition to the standard and extended controls, Microsoft has a Codeplex project (available with source code and tests, under the same open source license as IronPython) called *Silverlight Toolkit*.[8]

The toolkit is a collection of controls and utilities for Silverlight, including `TreeView`, `DockPanel`, and charting components.

USING THE EXTENDED CONTROLS

Visual Studio Tools for Silverlight comes with a lot more than just a new set of controls. It also includes additional assemblies for working with JSON, XML, and so on (don't use the outdated version of IronPython it includes, though!). What we're about to cover is just as relevant for using these other assemblies as it is for the extended controls.

The two assemblies that implement the extended controls are

- `System.Windows.Controls.dll`
- `System.Windows.Controls.Data.dll`

As with using other assemblies from IronPython, in order to use them we need to add a reference to them with `clr.AddReference`.

```
import clr
clr.AddReference('System.Windows.Controls')
```

Table 13.1 lists the Silverlight controls.

Table 13.1 Silverlight controls

Border	DatePicker	MultiScaleImage	Slider
Button	Grid	OpenFileDialog	StackPanel
Calendar	GridSplitter	Panel	TabControl
Canvas	HyperlinkButton	PasswordBox	TextBlock
CheckBox	Image	ProgressBar	TextBox
ComboBox	InkPresenter	RadioButton	ToggleButton

[7] Silverlight Tools for Visual Studio 2008 works with Visual Studio 2008 or Visual Web Developer 2008 Express. The tools are linked to from http://silverlight.net/GetStarted/default.aspx.

[8] See http://www.codeplex.com/Silverlight.

Table 13.1 Silverlight controls *(continued)*

ContentControl	ListBox	RepeatButton	ToolTip
DataGrid	MediaElement	ScrollViewer	UserControl

It will be self-evident what most of these controls are for from their names (one of the advantages of consistent naming schemes). One that may not be familiar is `MultiScaleImage`.[9] This control is for displaying multiresolution images using Deep Zoom. It allows the user to zoom and pan across the image.

These assemblies extend the `System.Windows.Controls` namespace, so the extended controls are imported from the same place as the standard controls. Of course, you can use the extended controls from XAML as well as from code.

EXTENDED CONTROLS FROM XAML

Using the extended controls from XAML requires us to tell the XAML loader where to find the classes that correspond to the XAML elements. We do this by adding extra `xmlns` declarations to the first tag in the XAML. Listing 13.6 is a part of an XAML file that uses several of the extended controls (`Calendar`, `DatePicker`, and `GridSplitter`).

Listing 13.6 XAML for a UI using the extended controls

```
<UserControl
   x:Class="System.Windows.Controls.UserControl"
   xmlns="http://schemas.microsoft.com/client/2007"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   xmlns:c="clr-
     namespace:System.Windows.Controls;assembly=System.Windows.Controls">
  <Grid>
     <Grid.RowDefinitions>
        <RowDefinition />
     </Grid.RowDefinitions>
     <Grid.ColumnDefinitions>
        <ColumnDefinition />
     </Grid.ColumnDefinitions>

    ...
      <StackPanel Grid.Row="0" Grid.Column="1"
       Margin="20,20,10,20">
        <ContentControl Content="ContentControl"
         Margin="5"/>
        <Button Content="ToolTipped Button" Margin="5"
         ToolTipService.ToolTip="Some ToolTip"/>
        <ToggleButton Content="ToggleButton" Margin="5"/>
        <c:DatePicker Margin="5"/>
        <c:Calendar Margin="5"/>
     </StackPanel>

     <c:GridSplitter Grid.Column="1"
      Width="5" HorizontalAlignment="Left"
      VerticalAlignment="Stretch"
```

[9] See http://msdn.microsoft.com/en-us/library/system.windows.controls.multiscaleimage(VS.95).aspx.

```
        ShowsPreview="True" />
    </Grid>
</UserControl>
```

To load this XAML from IronPython, like the example in listing 13.3,[10] we *don't* need to explicitly add references to the assemblies containing the controls; the XML namespace declarations do this for us. The `xmlns:c` declaration also declares a prefix (`c`) that we must use to reference controls from the `System.Windows.Controls` assembly. For example, the `DatePicker` control is referenced with `c:DatePicker` in the XAML.

Once this XAML is loaded, the results look like figure 13.5.

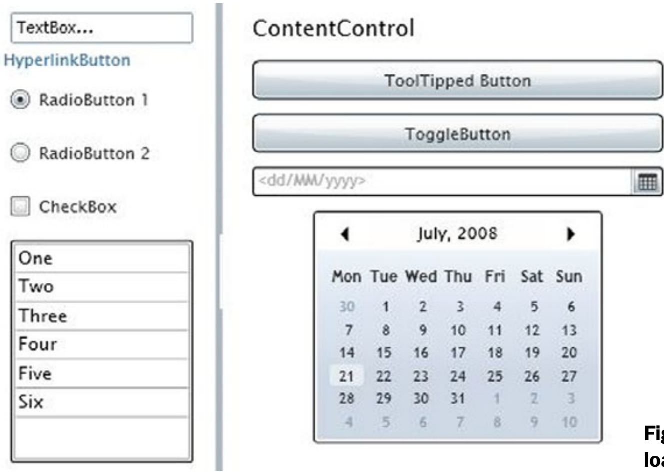


Figure 13.5 A user interface loaded from XAML

If you don't like the standard silver theme, you can use a new theme for all the controls in an application. In discussing the extended controls, there's an important point that we have glossed over.

13.1.4 Packaging a Silverlight application

If you look in the source code for the example applications that use the extended controls, you'll see a file that we haven't yet discussed, AppManifest.xaml. This is an XML file that tells Silverlight not only what assemblies your application uses but also which one provides the entry point. The reason it isn't included in the earlier examples is that Chiron can autogenerate it for applications that use no assemblies beyond the standard IronPython/DLR ones.

When you deploy a typical dynamic application, the xap file contains the following:

- app.py—your main Python file
- The IronPython assemblies (dlls)

---

[10] This XAML has a `UserControl` as the root element, so you'll need to modify listing 13.3 to create a `UserControl` instead of a `Canvas`. This is shown in ControlsExample2 in section 13.1.3 of the downloadable source code.

- An XAML (XML) manifest file
- Any additional Python modules, XAML files, assemblies, or resources your app uses

Listing 13.7 shows the manifest file needed for a basic dynamic application. You can either copy and paste this into your applications or let Chiron create it for you.

Listing 13.7 The XML manifest file for an IronPython application

```
<Deployment
 xmlns="http://schemas.microsoft.com/client/2007/deployment"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
 RuntimeVersion="2.0.31005.00"
 EntryPointAssembly="Microsoft.Scripting.Silverlight"
 EntryPointType="Microsoft.Scripting.Silverlight.DynamicApplication">
 <Deployment.Parts>
   <!-- Add additional assemblies here -->
   <AssemblyPart Source="Microsoft.Scripting.dll" />
   <AssemblyPart Source="Microsoft.Scripting.Core.dll" />
   <AssemblyPart Source="IronPython.dll" />
   <AssemblyPart Source="IronPython.Modules.dll" />
   <AssemblyPart Source="Microsoft.Scripting.ExtensionAttribute.dll" />
   <AssemblyPart Source="Microsoft.Scripting.Silverlight.dll" />
 </Deployment.Parts>
</Deployment>
```

Requiring XML manifest files may seem like unnecessary overhead for dynamic applications, but they serve a very serious purpose: making Silverlight applications fit for the enterprise.

It is interesting to note the `EntryPoint` attribute in the top-level `Deployment` element. This is how Silverlight knows how to execute a dynamic application, and in fact a dynamic application is just a normal C# application as far as Silverlight is concerned. Microsoft.Scripting.Silverlight.dll actually does the magic for us.

To use additional assemblies in your applications, including the extended controls, you need to include them in the manifest. This has a serious downside, even if your application is only a single Python file a few kilobytes in size; the resulting xap file will need to contain the IronPython assemblies and be several hundred kilobytes in size.

Fortunately, there is a way around this. If instead of just specifying the assembly name as the `Source` attribute for the assemblies, you specify an absolute URL, the assemblies will be fetched separately rather than being expected inside the xap file. If several of your applications share the same assemblies, then the browser can cache them, and your application can shrink back down to a more manageable size.[11]

We've covered all the basics now, and in fact you have everything you need to start writing Silverlight applications; everything else is mere detail. The best way of learning

[11] At some point a mechanism like the Global Assembly Cache will be implemented for Silverlight, but there isn't one yet.

these details is to use them, so in the next section we look at a more substantial Silverlight application: a Twitter client.

13.2 A Silverlight Twitter client

Deploying applications through the web has the massive advantage of *deploy once, use anywhere*, but with JavaScript it means coping with the pain of cross-browser issues and slow client-side performance. With Silverlight we can bring our structured application programming techniques and adapt them for the browser, often using the *same* libraries that we use on the desktop. In this section we look at a basic Silverlight Twitter client written in IronPython,[12] shown in figure 13.6.



Figure 13.6 An IronPython Silverlight Twitter client

This example consists of about 600 lines of code in total. We won't go through all the code line by line, but through it we can explore the following aspects of working with Silverlight:

- Cross-domain policies and tips for debugging Silverlight programs
- Creating user interfaces
- Making web requests from Silverlight
- Using XML from the Twitter API
- Threading and asynchronous callbacks and dispatching onto the UI thread
- Storing data in the browser with `IsolatedStorage`
- Timers

In some ways a Twitter client is a difficult example for Twitter. It means working with data fetched from an external server, and we quickly run into problems with making cross-domain calls from Silverlight. We start by looking at what you can and can't do.

13.2.1 Cross-domain policies

When writing web client applications with JavaScript you can make calls back to the server with `XMLHttpRequest`, or its equivalent. This uses cookies and authentication that the browser has cached for the site you are accessing. If JavaScript on a web page

[12] The user interface is deliberately gaudy so that you can visually see the way the UI elements are nested inside each other.

could access any domain, this would be a security hole, as applications could access any site effectively logged in as you. To prevent this, the browser restricts JavaScript to allow only requests to the same domain as the current web page, blocking cross-domain calls.[13]

In theory, Silverlight doesn't have the same problems as JavaScript (it does use the browser networking stack under the hood but works with it directly). However, for nebulous security reasons, Silverlight still applies some restrictions. Silverlight allows your applications to make *some* cross-domain calls, so long as the domain you are accessing allows it.

The way a domain allows calls from a Silverlight application is by providing a clientaccesspolicy.xml file at the top level of the domain.[14] Listing 13.8 is an example client-access policy file that allows access to the whole domain and from any referring domain.

Listing 13.8 A clientaccesspolicy.xml file to allow cross-domain calls into a website

```
<?xml version="1.0" encoding="utf-8"?>
<access-policy>
  <cross-domain-access>
    <policy>
      <allow-from http-request-headers="*">      <- Allow POST requests
        <domain uri="*"/>                        <- Specify domains allowed access
      </allow-from>
      <grant-to>
        <resource path="/" include-subpaths="true"/>   <- Specify which paths are available
      </grant-to>
    </policy>
  </cross-domain-access>
</access-policy>
```

Of course, this is most useful when the domain you want to access is under your control. If it isn't your server, the typical solution is to proxy the service you want to access. This means that your application queries a server that *is* under your control, and the server makes the query and returns the result.

Unfortunately, although Twitter exposes a nice and simple API for clients to use, it doesn't have the necessary client-access policy allowing us to query it from Silverlight.[15] To make the client work, we've implemented a simple Python proxy server that runs locally. Because we will be using Chiron, which serves on port 2060, the local proxy server runs on port 9981. You can start it by executing the following command from the directory containing the example code for this section:

```
python simple_proxy.py
```

---

[13] There are various ways around this restriction from JavaScript, of course. For example, the jQuery library provides an API for accessing web services across domains.

[14] Silverlight also supports a subset of the crossdomain.xml schema used by Flash. See the following page for details: http://msdn.microsoft.com/en-us/library/cc197955(VS.95).aspx.

[15] I (Michael) *really* wanted to write a Twitter client though, as I've been having a lot of fun with it recently. You can follow me at http://www.twitter.com/voidspace.

As far as Silverlight is concerned, calls to this server are still cross domain, so our proxy server has to provide a valid client access policy when /clientaccesspolicy.xml is requested. Before we look at the substance of the application, let us share a tip for debugging Silverlight applications.

13.2.2 Debugging Silverlight applications

The normal first resort for tracking down bugs in Python programs is the judicious use of print statements. This technique doesn't work in Silverlight, and to make it worse, the error messages can be extremely cryptic.[16] There is a way around this, though. Python allows us to override standard output with a custom writer. Listing 13.9 shows how to create a custom writer and set it on `sys.stdout`.[17]

Listing 13.9 Diverting standard out to an HTML text area

```
import sys
from System.Windows.Browser import HtmlPage

class Writer(object):
   def __init__(self):
      self.stdout = ''

   def write(self, text):
      self.stdout += text
      element = HtmlPage.Document.debugging
      element.value = self.stdout

output_writer = Writer()
sys.stdout = output_writer
```

- `element = HtmlPage.Document.debugging` — **The textarea HTML element**
- `element.value = self.stdout` — **Set the text**
- `sys.stdout = output_writer` — **Divert standard out**

When a print statement is executed, the `write` method of our custom writer is called. This looks up the HTML element with the id `debugging` and sets the text on it. We fetch this element from the `HtmlDocument`,[18] which we get hold of via `System.Windows.Browser.HtmlPage.Document`. Figure 13.7 shows the `textarea` with the debugging output from running the Twitter client.

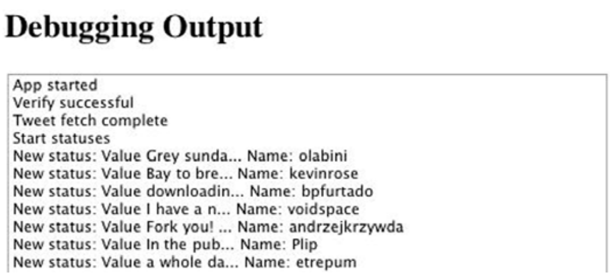


Figure 13.7 The HTML `textarea` containing the output of print statements from the Twitter client

---

[16] This was to save space in the Silverlight runtime and may be improved in a future version of Silverlight.

[17] The actual code in the example is *slightly* different. We shouldn't modify the browser DOM from anything other than the UI thread, so it contains code to dispatch the `write` in case we want to print from an asynchronous callback off the main thread. This topic is covered in more detail later in the chapter.

[18] See http://msdn.microsoft.com/en-us/library/system.windows.browser.htmldocument(VS.95).aspx.

Silverlight provides other ways to interact with the browser DOM, which we look at shortly. First we look at parts of the Silverlight UI model that we haven't already seen.

13.2.3 The user interface

The main user interface for the Twitter client is a `Border` containing a `StackPanel`, two classes we worked with in the chapter on WPF. The main application is contained in a class called `MainPanel` that inherits from `StackPanel`. The rest of the user interface is laid out by nesting `StackPanels` in new borders where necessary. From the wonderful color scheme I chose, you should be able to see the different elements nested inside each other.

When you first log in, you are presented with the login panel shown in figure 13.8.

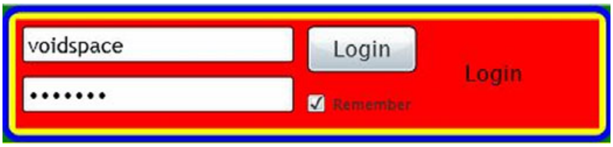


Figure 13.8 The Silverlight Twitter client login panel

This is a horizontally oriented `StackPanel` (in a border) with nested vertical `StackPanels` containing the username/password textboxes, the login button, a check box, and a textblock for messages.

We haven't yet used the `CheckBox`, but like the other user interface components, it's very simple. Listing 13.10 shows the configuration of the login button and the Remember me check box in the login panel.

Listing 13.10 Configuring the `Button` and `CheckBox` in the login panel

```
from System.Windows import Thickness, HorizontalAlignment

from System.Windows.Controls import (
   Button, StackPanel, CheckBox
)

button = Button()
button.Content = '  Login  '
button.FontSize = 16
button.Margin = Thickness(5, 5, 5, 5)
stretch = HorizontalAlignment.Stretch
button.HorizontalAlignment = stretch          <- Set the button to stretch horizontally

remember_me = CheckBox()      <- Create the CheckBox
remember_me.IsChecked = True                   <- Check it
remember_me.Margin = Thickness(5, 5, 5, 5)
remember_me.Content = 'Remember'

button_pane = StackPanel()
button_pane.Children.Add(button)
button_pane.Children.Add(remember_me)
```

The textbox for the username is a straightforward `TextBox`, but we can't use that for entering the password because we don't want it to be visible while it is being typed.

Thankfully, a password textbox is one of the standard controls. The difference between the `PasswordBox` API and a straight `TextBox` is that instead of setting and fetching the `Text` property, we use the `Password` property.

Once the user has logged in, the tweets are fetched from the Twitter API and displayed inside a grid.

THE GRID, THE SCROLLVIEWER, AND THE HYPERLINKBUTTON

The grid is inside a `ScrollViewer`, so that all the Twitter messages can be seen. The grid has two columns. The Twitter usernames are displayed in the left column as clickable links, with the Twitter messages in the right column (these are all visible in figure 13.6). The code that does all this is in listing 13.11.

Listing 13.11 Creating and populating the main grid for the Twitter client

```
from System.Windows import (
   HorizontalAlignment, GridLength,
   VerticalAlignment, TextWrapping,
   FontWeights
)

from System.Windows.Controls import (
   ScrollViewer, ScrollBarVisibility,
   TextBlock, Grid, HyperlinkButton,
   ColumnDefinition, RowDefinition,
)

from System.Windows.Media import Colors, SolidColorBrush
from System import Uri

viewer = ScrollViewer()
viewer.Height = 475
viewer.Background = SolidColorBrush(Colors.White)
auto = ScrollBarVisibility.Auto
viewer.VerticalScrollBarVisibility = auto          <- Make the scrollbar visible when needed

grid = Grid()
grid.ShowGridLines = True
viewer.Content = grid

first_column = ColumnDefinition()
first_column.Width = GridLength(115.0)             <- Set explicit width on left column
grid.ColumnDefinitions.Add(first_column)
second_column = ColumnDefinition()
grid.ColumnDefinitions.Add(second_column)

for i in range(len(statuses)):
   grid.RowDefinitions.Add(RowDefinition())        <- One row per message

def configure_hyperlink(block, col, row):          <- Function to configure the HyperlinkButton
   block.FontSize = 14
   block.Content = name
   uri = Uri('http://twitter.com/%s' % name)
   block.NavigateUri = uri                         <- Set the link on the HyperlinkButton
   block.FontWeight = FontWeights.Bold

   block.HorizontalAlignment = HorizontalAlignment.Left
   block.VerticalAlignment = VerticalAlignment.Center
```

```
    grid.SetRow(block, row)
    grid.SetColumn(block, col)

for row, status in enumerate(statuses):
    name = status['name']
    text = status['text']

    block1 = HyperlinkButton()
    configure_hyperlink(block1, 0, row, name)

    block2 = TextBlock()
    block2.Text = text
    block2.TextWrapping = TextWrapping.Wrap     <-- Wrap the text in the TextBlock
    block2.FontSize = 14
    grid.SetRow(block2, row)
    grid.SetColumn(block2, 1)

    grid.Children.Add(block1)
    grid.Children.Add(block2)
```

There are several details worth noticing in this code. Using color brushes in Silverlight is slightly different from in WPF. Instead of using something like `Brushes.White` to set the background, we construct a brush with `SolidColorBrush(Colors.White)`.

The `HyperlinkButton` is placed at the left and the center (vertically) by setting horizontal and vertical alignment. By default the `HyperlinkButton` opens in the same window. You can specify that the link should open in a new window by setting `HyperlinkButton.TargetName = "_blank"`.[19]

Because the Twitter messages are often longer than the width of the grid, we need to ensure that the `TextBlock` wraps the text by setting the `TextWrapping` property. We've covered the important parts of the user interface, so now we look at how we access server resources from Silverlight.

13.2.4 Accessing network resources

The basic class for accessing network resources from Silverlight is the `WebClient`, which can be used for both GET and POST requests. It has an asynchronous API, so you configure it to fetch an API and provide callbacks to handle the response.

THE WEBCLIENT

The Twitter API is extremely easy to work with,[20] by virtue of its *XML or JSON over REST* interface. To verify login details or fetch the latest messages for a Twitter user, you simply fetch a URL from the Twitter API (using basic authentication with the user credentials), which then returns either XML or JSON, depending on which format you asked for in the URL.

Because of the cross-domain policy, we can't access the Twitter domain, so we can leave the proxy server to handle the authentication. The `twitter_proxy` module wraps the `WebClient` in a `Fetcher` class, shown in listing 13.12, which takes a callback function along with the username, the password, and the action to be performed.

---

[19] Although currently that stops the link button working altogether on Safari on the Mac!
[20] See http://groups.google.com/group/twitter-development-talk/web/api-documentation.

Listing 13.12 The `Fetcher` class for downloading web resources with `WebClient`

```
from System import Uri
from System.Net import WebClient
from System.Windows.Browser import HttpUtility

get_url = 'http://localhost:9981/?action=%s&name=%s&pass=%s'

class Fetcher(object):
   def __init__(self, name, password, action, callback):
      name = HttpUtility.UrlEncode(name)                    <- UrlEncode values for the GET string
      password = HttpUtility.UrlEncode(password)
      uri = Uri(get_url % (action, name, password))
      self.callback = callback

      web = WebClient()

      web.DownloadStringCompleted += self.completed         <- Hook up async callback event
      web.DownloadStringAsync(uri)                          <- Start the request

   def completed(self, sender, event):
      if not event.Error or event.Cancelled:
         self.callback(event.Result)
```

It is called like this:

```
Fetcher(username, password, 'fetch', callback)
```

If you watch the local proxy for the first time this is called, you will see that before fetching the requested URL, Silverlight asks for /clientaccesspolicy.xml. If the server doesn't respond to this, then an exception will be raised.

WebClient and changing data

Under the hood, `WebClient` caches requests for us. If we fetch the same URL later, even from a new instance, the same data will be returned instead of a new request being made. This is a problem if the data you want changes.

In the Silverlight Twitter client we solved this by adding a digit to the end of the URL sent to the proxy. The digit is incremented with every request so that `WebClient` sees a different URL every time.

Because we are putting the action, username, and password as parameters into the GET string, they need to be URL encoded. This is the task of the `HttpUtility` class, which also has a corresponding `UrlDecode` method that could be useful if you ever want to look at the query parameters of the current URL.[21]

When the response is available, the `DownloadStringCompleted` event fires and the `completed` method is called. If the request completes successfully, then our original callback is called with the response as a string.

[21] This is made available through `System.Windows.Browser.HtmlPage.Document.DocumentUri`.

PARSING THE XML FROM TWITTER

The main client application takes the results (a string of XML) and needs to parse this into messages suitable for populating the grid. Because Silverlight comes with the `XmlReader` (and the associated classes and enumerations), we can actually reuse the `XmlDocumentReader` class from chapter 5 to parse the response.

Literally the only changes we needed to make to this class were to add a reference to the `System.Xml` assembly and the addition of a single line to handle the XML declaration that we didn't need for `MultiDoc`. Since we have our XML as a string, we can pass a `TextReader` into the call to `XmlReader.Create` instead of a stream. Just as in the desktop .NET framework, `XmlReader` lives in the `System.Xml` assembly, so we need to add a reference to it before we can import from it.

The XML that Twitter returns when we ask for a user's messages has a top-level `statuses` element containing a series of `status` blocks nested in it. Each one of these has various elements that describe the message and the user who posted it. For our basic client, we are interested in only the body of the message itself and the name of the user who posted it. With the event-based parsing of `XmlReader`, we need only define handler methods for the parts of the Twitter XML that we are using, ignoring the rest. You can see the (brief) code that does this in the `twitter` module. It returns a list of dictionaries, each dictionary with name and text members, ready for displaying in the grid.

POSTING WITH WEBCLIENT

For posting a Twitter message, we need to make a POST request. We can also use the `WebClient` for this, using the `UploadStringAsync` method instead. Listing 13.13 shows the `Poster` class, which posts Twitter messages.

Listing 13.13 Making POST requests with `HttpWebRequest`

```
from System importUri
from System.Net import WebClient
from System.Windows.Browser import HttpUtility

class Poster(object):

    def __init__(self, tweet, username, password, callback):
        uri = Uri(post_url)
        self.callback = callback

        print 'Tweeting:', tweet[:10]
        tweet = tweet[:140]                      <- Shorten the message to maximum 140 characters

        url_encoded_tweet = HttpUtility.UrlEncode(tweet)      <- URL-encode Twitter message

        username = HttpUtility.UrlEncode(username)
        password = HttpUtility.UrlEncode(password)

        data = 'username=%s&password=%s&tweet=%s' % (username,
              password, url_encoded_tweet)                    <- Create POST data

        web = WebClient()
        web.UploadStringCompleted += self.completed           <- Hook up callback for completion
        web.UploadStringAsync(uri, "Post", data)              <- Make the POST
```

```
    def completed(self, sender, event):
        if event.Cancelled or event.Error:
            # Post failed
            print 'POST failed'
            return self.callback('')
        return self.callback(event.Result)
```

This Poster class wraps up the API, so that when we instantiate Poster with the data for the post and a callback function, it will make the request, and the callback will be called with the results once the request is complete.

> **Posting to the proxy**
>
> Since the only time we make POST requests to the proxy server it is to post a new message, there is no need to specify the action in the URL. When I originally implemented this, I set the post URL to be http://localhost:9981 and then spent a *long* time trying to find out why it didn't work.
>
> Of course, the client access policy specifies that all URLs below / are allowed, so changing the post URL to http://localhost:9981/ (with the trailing slash) worked!

As is common in .NET, asynchronous callbacks often happen on a different thread than the one you create them from, and we need to handle this.

13.2.5 Threads and dispatching onto the UI thread

Like WPF and Windows Forms, the Silverlight user interface runs within a single main thread. Any operations that interact with user interface elements, or access the browser DOM, should be done from this thread.

Fortunately this is straightforward. WPF uses dispatchers to invoke delegates onto the UI thread, and Silverlight includes a cut-down version of this system. Silverlight provides a single dispatcher, which is accessible via the Dispatcher property on user interface components. From IronPython, functions that we pass into the Dispatcher.BeginInvoke method are invoked onto the main thread.

A lot of the core threading classes from .NET are available in Silverlight. Listing 13.14 creates a new thread, and after a brief pause it uses the dispatcher to invoke a function that changes the text on a textblock.

Listing 13.14 Using the dispatcher to modify the user interface from another thread

```
from System.Windows import Application
from System.Windows.Controls import Canvas, TextBlock
from System.Threading import Thread, ThreadStart

root = Canvas()
Application.Current.RootVisual = root

text = TextBlock()
thread_id = Thread.CurrentThread.ManagedThreadId
```

```
text.Text = "Created on thread %s" % thread_id
text.FontSize = 24
root.Children.Add(text)

def wait():
    Thread.Sleep(3000)                                    # Pause for three seconds
    thread_id = Thread.CurrentThread.ManagedThreadId
    def SetText():
        text.Text = 'Hello from thread %s' % thread_id
    text.Dispatcher.BeginInvoke(SetText)                  # Invoke onto the UI thread

t = Thread(ThreadStart(wait))      # Create a new thread
t.Start()
```

When this code is run, the textblock displays the message "Created on thread 1." After three seconds, this changes to show the thread ID of the new thread that we created.

This is simple enough, but there might not always be a convenient user interface element available to dispatch on from code that needs it. The Twitter client has a `dispatcher` module that does this for us. When the UI is first created, the main application (in app.py) calls the function `SetDispatcher`. The `dispatcher` module also exports two other functions, `Dispatch` and `GetDispatchFunction`. These either immediately dispatch a function for us or turn a function into one that is dispatched when it is called (which is useful for creating dispatched callbacks). `Dispatch` and `GetDispatchFunction` are used throughout the Twitter client, where code might interact with the user interface.

A common need is to have some event regularly occur at a timed interval. We use this in the Twitter client to fetch the latest tweets every 60 seconds. We can do this with the `DispatcherTimer`. Listing 13.15 uses a `DispatcherTimer` to update a counter in a textblock.

Listing 13.15 Using a `DispatcherTimer` for timed events on the UI thread

```
from System.Windows.Threading import DispatcherTimer
from System import TimeSpan

text = TextBlock()
text.Text = "Nothing yet"
text.FontSize = 24
root.Children.Add(text)

counter = 0
def callback(sender, event):
    global counter
    counter += 1
    text.Text = 'Tick %s' % counter

timer = DispatcherTimer()
timer.Tick += callback
timer.Interval = TimeSpan.FromSeconds(2)
timer.Start()
```

When this code is executed, the `Tick` event fires every two seconds and increments the counter. Because `Tick` is executed on the UI thread, there is no need for us to

explicitly dispatch the callback. If we didn't need to interact with the user interface, then we could use the `System.Threading.Timer` class instead.

Another of the Silverlight APIs that the Twitter client uses is the isolated storage system, for storing user login details in the browser.

13.2.6 IsolatedStorage in the browser

Isolated storage provides a mechanism for applications to store data in the browser cache. The intent of this is to provide a temporary cache for applications or for storing configuration information. Data stored there does persist but is destroyed if the user clears the browser cache. With this in mind, the default limit *per application* is 100 kilobytes of storage. You can request more, which will present the user with a dialog requesting permission to increase the limit for this application.

Isolated storage provides a filesystem-like mechanism that we access through the `System.IO.IsolatedStorage` namespace. We can list the files (or directories) stored and load, save, or delete files.

In the Twitter client this is used for the user login credentials. If the Remember check box is checked when the user logs in, then the username and password are saved in a file.[22]

When the application is first loaded, if that file exists in the application storage, then the file is loaded and the username and password textboxes are populated from it.

The basis of using isolated storage from our applications is to get a data store by calling `IsolatedStorageFile.GetUserStoreForApplication()`. This returns the `IsolatedStorageFile` instance for the current application. Listing 13.16 shows three functions to load and save files listed in the data store and to list the contents.

Listing 13.16 Using the Silverlight isolated storage

```
from System.IO.IsolatedStorage import (
   IsolatedStorageFile, IsolatedStorageFileStream
)
from System.IO import FileMode, StreamReader, StreamWriter

def ListFiles():
   store = IsolatedStorageFile.GetUserStoreForApplication()
   return store.GetFileNames('.')                            <- List all files in the store

def DeleteFile(name):
   store = IsolatedStorageFile.GetUserStoreForApplication()
   store.DeleteFile(name)

def SaveFile(name, data):
   store = IsolatedStorageFile.GetUserStoreForApplication()
   mode = FileMode.Create                                    <- Mode to write files
   iStream = IsolatedStorageFileStream(name, mode, store)

   writer = StreamWriter(iStream)
```

[22] In a plain text file, but this is not recommended for production systems storing sensitive user data like passwords!

```
    writer.Write(data)                              Write into the
                                                    data store
    writer.Close()
    iStream.Close()

def LoadFile(name):
    store = IsolatedStorageFile.GetUserStoreForApplication()     Mode to
    mode = FileMode.Open                                         read files
    iStream = IsolatedStorageFileStream(name, mode, store)

    reader = StreamReader(iStream)
    data = reader.ReadToEnd()                       Read from the
                                                    stored file
    reader.Close()
    isolatedStream.Close()

    return data
```

The data store *isn't* a flat file system; we can create subdirectories and work with those as well. The store has many useful methods;[23] one of the more important ones is `TryIncreaseQuotaTo` to request an increase in the amount of storage available. This method can be called *only* from inside the event handler of a control such as a button. You pass in the amount of space you want (in bytes), which presents a dialog to the user to approve the request. The method returns a Boolean indicating whether the request succeeded or not. The following snippet of code shows a request to double the amount of storage for an application:

```
from System.IO.IsolatedStorage import IsolatedStorageFile
store = IsolatedStorageFile.GetUserStoreForApplication()
space = store.AvailableFreeSpace
success = store.TryIncreaseQuotaTo(space * 2)
```

Figure 13.9 shows the dialog presented to a user when this code runs on Safari under Mac OS X.

If you attempt to execute this code outside a user-interface event handler, then it will fail and return `False` without showing the dialog.



Figure 13.9 Requesting to increase the storage for a Silverlight application

We've now seen that Silverlight contains all the necessary ingredients for creating serious applications that live on the web. Writing web applications is different from programming for the desktop. Although the CoreCLR lessens that difference, the key to creating effective applications is understanding the difference, and that means being able to make good use of the important Silverlight APIs such as `IsolatedStorage`.

There are a couple more important Silverlight APIs that we have only skirted around the edges of; these are working with videos and the browser DOM.

[23] These are listed at http://msdn.microsoft.com/en-us/library/system.io.isolatedstorage.isolatedstoragefile_members(VS.95).aspx.

13.3 Videos and the browser DOM

Despite its diminutive download, Silverlight packs in a great deal. The intended use for Silverlight spans the gamut of games, applications, and media streaming, so along with the .NET Base Class Libraries it comes with APIs specific to these tasks. Some of these are the adapted version of their complementary WPF class in the desktop framework. Others, like those for working with the browser DOM, are new to Silverlight. In this final section of the chapter, we use two more of the core Silverlight APIs.

13.3.1 The MediaElement video player

The media capabilities are provided in part through the `MediaElement` class. Figure 13.10 shows a video as part of a Silverlight canvas.

This class is a control, and like the other controls, it lives in the `System.Windows.Controls` namespace. Having instantiated it, you specify a data source as a `Uri`, as in listing 13.17.



Figure 13.10 The `MediaElement` class in action

Listing 13.17 Using the `MediaElement` video control

```
from System.Windows.Controls import MediaElement
from System import Uri, UriKind

video = MediaElement()                                  <- Create the MediaElement
source = Uri('SomeVideo.wmv', UriKind.Relative)        <- The Uri for the video file
video.Volume =                                          <- A number from 0-l
video.Source = source
video.Width = 450                                       <- Scale the video
```

As always, the `MediaElement` has methods, properties, and events that we haven't used here. The methods to start and stop playing are `Play` and `Pause`, but the video will start playing as soon as it has downloaded events, so it is unnecessary. Other useful properties are `Position`, which we can set with a `TimeSpan` object, and both `Width` and `Height` to scale the video. As with setting the `Width` and `Height` scales, setting one automatically adjusts the other.

In addition to using it directly, we can use the `MediaElement` as the source for a `VideoBrush`, which we can use as the foreground mask on another control or to fill a shape, which we then transform or animate. Listing 13.18 shows how to set a video as the foreground mask for the text in a textblock.

Listing 13.18 Setting a `VideoBrush` with a video on a `TextBlock`

```
from System import TimeSpan, Uri, UriKind
from System.Windows import Application, RoutedEventHandler
from System.Windows.Controls import (
```

```
    Canvas, TextBlock, MediaElement
)
from System.Windows.Media import VideoBrush, Stretch

root = Canvas()

source = Uri('../SomeVideo.wmv', UriKind.Relative)
video = MediaElement()
video.Source = source
video.Opacity = 0.0
video.IsMuted = True

def restart(s, e):
    video.Position = TimeSpan(0)
    video.Play()

video.MediaEnded += restart

brush = VideoBrush()
brush.Stretch = Stretch.UniformToFill
brush.SetSource(video)

t = TextBlock()
t.Text = 'Video'
t.FontSize = 120
t.Foreground = brush

root.Children.Add(t)
root.Children.Add(video)

Application.Current.RootVisual = root
```

Make the MediaElement itself invisible

Silence is golden

Set the video to repeat

The fill style for the brush

Add the video as well[24]

MediaElement exposes an event called MediaEnded that fires when the video ends. We make the video loop by hooking up a restart function to this event that restarts the video.

Figure 13.11 shows the results of setting a VideoBrush on a TextBlock from IronPython.

An important aspect of Silverlight that we have touched on only briefly is working with the browser and the Document Object Model.



Figure 13.11 A VideoBrush showing through the text on a TextBlock

13.3.2 Accessing the browser DOM

We flirted with the DOM when we looked at diverting standard output, so that debugging print statements would appear in an HTML text area. We access HTML elements in the page through the Document property on System.Windows.Browser.HtmlPage. We can access elements by using their id as the attribute name on Document, which does a dynamic lookup (one of the joys of working with a dynamic language). This returns an element object,[25] with which we can do many useful things.

[24] In my initial experiments, I forgot to add the MediaElement as well and then spent an hour trying to work out why it wasn't working.

[25] Specifically, an HtmlElement object. See http://msdn.microsoft.com/en-us/library/system.windows.browser.htmlelement(VS.95).aspx.

Listing 13.19 shows how to set the `innerHTML` on an element, using `GetElementById` as an alternative way to fetch elements, modifying style rules on an element with `SetStyleAttribute`, plus setting properties with the `SetProperty` method.

Listing 13.19 Interacting with DOM elements from inside Silverlight

```
from System.Windows.Browser import HtmlPage

document = HtmlPage.Document
element = document.some_element
element.innerHTML = "Some <em>HTML</em>."

element2 = document.GetElementById('another_element')
element2.SetStyleAttribute('border', 'solid black 2px')

element3 = document.GetElementById('some_textbox')
element3.SetProperty('disabled', True)
element3.value = 'Text in a textblock'
```

As well as using `innerHTML` we could also use `innerText`. Because we have access to these attributes, we can use all our favorite AJAX tricks from Silverlight.

Some properties on HTML elements require us to use the `GetProperty` and `SetProperty` methods rather than simple attribute access. Fetching the `disabled` property of a textblock as an attribute will return a string instead of a Boolean (and setting it directly with a string *or* a Boolean doesn't work), so you should use the getter and setter methods instead.

In addition to access to the HTML elements, we can also set event handlers and interact with JavaScript. Listing 13.20 does three things:

1. Adds a mythical `DoSomething` Python function as an event handler for a button in the HTML with the id `some_button`.
2. Calls a JavaScript function (`function_name`) with a string argument.
3. Creates a JavaScript object (`XmlHttpRequest`) and invokes methods on it (which executes a synchronous GET request JavaScript and allows us to retrieve result).

Listing 13.20 DOM events and JavaScript from Silverlight

```
from System import EventHandler

handler = EventHandler(lambda sender, event: DoSomething())   # Create an EventHandler delegate
button = document.some_button
button.AttachEvent(handler)

jscriptfunc = "function_name"
argument = "Something"
HtmlPage.Window.CreateInstance(jscriptfunc, argument)

request = HtmlPage.Window.CreateInstance("XMLHttpRequest")
request.Invoke("open", "GET", url, False)   # False makes the request synchronous
request.Invoke("send", "")
result = request.GetProperty("responseText")
```

These simple tricks make it possible to create hybrid applications with both JavaScript *and* Silverlight, which communicate with each other.

There is one final way of interacting with the browser that we need to mention, and it is a bit more indirect. If we give the Silverlight control an id when we embed it in HTML, we can access it in the same way we have accessed any other element by id. A better way to get a reference to the current Silverlight control is

```
System.Windows.Application.Current.Host
```

The `SilverlightHost` object itself is not particularly interesting, but it exposes a `Content` subobject that is. This has members[26] like `ActualHeight` and `ActualWidth` that tell us the real size of the Silverlight control within the web page. `IsFullScreen` will not only tell us if the Silverlight control is operating in full-screen mode, but if set to `True` from a button event handler it will also switch the plugin to full-screen mode. Most important, though, the content object also has a `ReSized` event, which fires when the control changes size. If, instead of creating the Silverlight control with a fixed size in the HTML embedding code, we let the browser size it, then we can respond appropriately (perhaps re–lay out the UI) when the control is resized.

We've had only one chapter to learn about Silverlight, but it's clear that this framework has a huge amount of potential. It's particularly exciting that programming it from IronPython is such a good experience; long may the reign of dynamic languages on the web continue.

13.4 Summary

Rich internet applications are already an important part of the internet revolution, and they're only becoming more important. Silverlight is one of the new frameworks that allow programmers to really take advantage of client-side processing power in web applications.

One of the tools we used when writing this chapter was the Silverlight IronPython Web IDE.[27] This allows us to experiment with the Silverlight APIs by executing code in the browser, and it has several examples preloaded, including some topics we didn't have space for here.

There's a lot we haven't had time to use. We haven't looked at loading XAML or initializing controls from XAML or animations with the `StoryBoard`. Because the Silverlight user interface model is based on WPF, these features are very similar to using them from WPF, and this includes using Expression Blend[28] as a design tool. With clever structuring, your desktop and online versions of your applications could share a lot of their code and XAML.

This is one of the best things about working with Silverlight; much of our existing knowledge about .NET and IronPython is directly applicable. This even includes extending IronPython with C# and embedding IronPython into C# or VB.NET applications, which are the focus of the next part of this book.

---

[26] See http://msdn.microsoft.com/en-us/library/system.windows.interop.content_members(VS.95).aspx.
[27] This is available online at http://www.voidspace.org.uk/ironpython/webide/webide.html.
[28] To use Expression Blend with Silverlight you'll need version 2.5 or later.

Part 4

Reaching out with IronPython

IronPython is just one of the now-numerous languages that run on the .NET framework. In the book so far we have been focusing on the different ways that we can access features of the .NET framework from within the IronPython world-view. In this final part of the book, we look at the other side of the story, working with IronPython from other .NET languages. We start this look by creating class libraries with C# and VB.NET specifically to use from IronPython.

14 Extending IronPython with C#/VB.NET

This chapter covers

- Creating class libraries for IronPython
- .NET attributes and IronPython
- Calling unmanaged code with P/Invoke
- Creating dynamic objects with C#/VB.NET
- Compiling assemblies at runtime

There is a standard mantra about performance when programming with Python (CPython). The mantra goes something like this: "Code in Python first; then profile and optimize your bottlenecks. Finally, if you still need your code to run faster, rewrite the performance-sensitive parts of your code in C." Okay, as mantras go there are probably snappier ones, but it still contains a lot of wisdom. Of course, with IronPython we wouldn't drop down to C[1] but into C# or VB.NET instead.

[1] Well, not usually. There are tools like SWIG that will allow you to generate .NET wrappers for C/C++ libraries.

This is an area where IronPython has a great advantage over CPython. Both C# and VB.NET are modern object-oriented languages with garbage collection and all the features of the common language runtime, and both are substantially easier to work with than C. In order to write a Python extension in C, you have to manage your own memory and use the Python C API to create and interact with Python objects. By way of contrast, we have shown how simple it is to use .NET classes from IronPython, and this includes classes you write yourself.

If you have come to IronPython from C# or VB.NET, then the parts of this chapter showing the syntax of these languages will already be familiar to you.[2] (Of course, if you're coming to IronPython from Python, this material will be a nice introduction to both of them.) The important thing to take from this chapter, though, is that it is possible to create objects that behave as you would expect in Python. This shouldn't really come as much of a surprise; after all, the native IronPython types like lists and dictionaries are written in C#. IronPython does lots of clever magic for us that allows us to use the standard .NET interfaces and mechanisms in ways that feel entirely natural to the Python programmer. In this chapter you will be learning how to take advantage of that magic.

Creating class libraries with Visual Studio and using them from IronPython is something we've already looked at. In chapter 6 we used the Visual Studio Windows Forms designer to create a dialog as a C# class library and then imported it from IronPython. In this chapter we take this further and look at writing class libraries in C# or VB.NET for use from IronPython. Specifically you'll be learning how to expose Python-friendly APIs from our .NET classes and even making them behave like dynamic Python objects. We finish off by dynamically compiling and using these class libraries at runtime.

14.1 Writing a class library for IronPython

Writing code in C# or VB.NET (or Boo or F# or any of the wealth of .NET languages that exist now) and using it from IronPython is straightforward. Improving performance is not the only reason to use an alternative language; in fact, my experience has been that IronPython is usually fast enough. It may be that you are writing a .NET class library and simply want to know how to make it as usable as possible from IronPython as well as other languages. Alternatively, you may be using C# to access features of the .NET framework that are hard to use from IronPython, like Linq or .NET attributes. In this section we write general-purpose class libraries and see how standard .NET concepts seamlessly map to their equivalents in Python. We also use C#/VB.NET to overcome some of the limitations of IronPython.

The first step is finding an IDE you are happy with. Actually, any text editor and the command line would do fine, and for Python development that's exactly what many programmers use. For statically typed languages, the need to invoke the compiler on many interdependent files in potentially large projects is at least *one* of the reasons

[2] And if not, there is an excellent appendix with an introduction to C#.

why most developers will use a full IDE. For Windows, Visual Studio (in one of its many variants) is a logical, but not the only, choice. For platforms *other* than Windows,[3] MonoDevelop makes a great development environment.

14.1.1 Working with Visual Studio or MonoDevelop

Visual Studio Express is free to download and use, but you have to choose which language you'll be developing with. There are separate versions for C#, C++, VB.NET, and web development, but you can have several different versions of Visual Studio Express installed on the same machine. If you have Visual Studio Professional, then you don't have to make a choice, as it supports all of the standard .NET languages. Figure 14.1 shows a new class library being created with the VB.NET version of Visual Studio Express.

MonoDevelop is an IDE written with the GNOME user-interface toolkit, and it works with Linux or Mac OS X.[4] It supports C, C#, and VB.NET and is included in the Mono package for Mac OS X. Figure 14.2 shows MonoDevelop in use editing a C# project.

Let's use these IDEs to create objects we can use from IronPython.

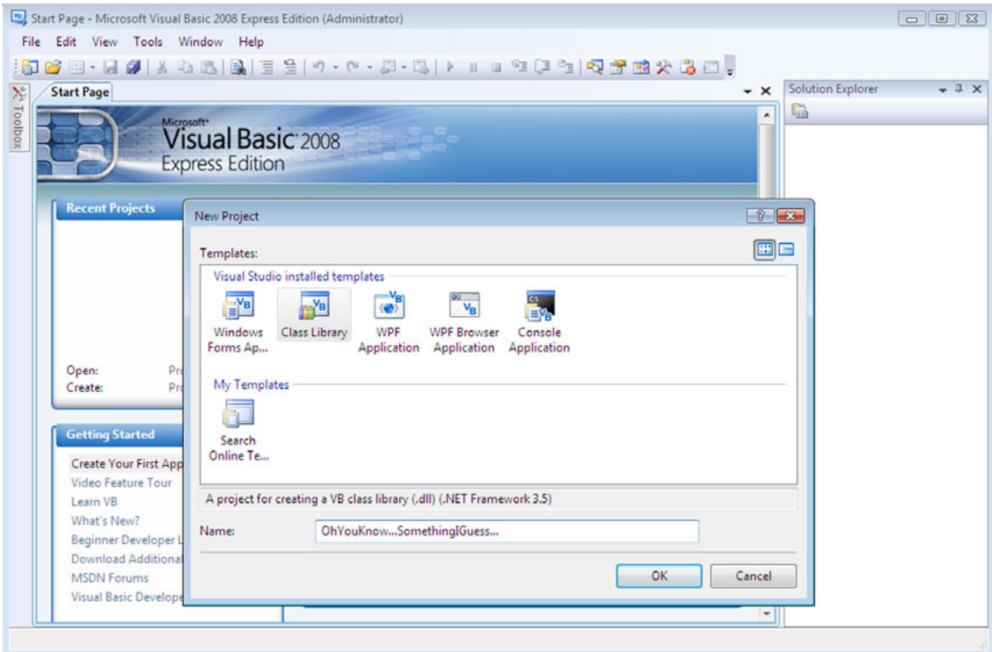


Figure 14.1 The VB.NET version of Visual Studio Express

---

[3] It is technically possible but very difficult to get MonoDevelop working on Windows. SharpDevelop is a good open source .NET IDE for Windows.

[4] It is *possible* to compile MonoDevelop for Windows, but it has a *lot* of dependencies. It is likely that prebuilt binaries for Windows will be made available as MonoDevelop matures.

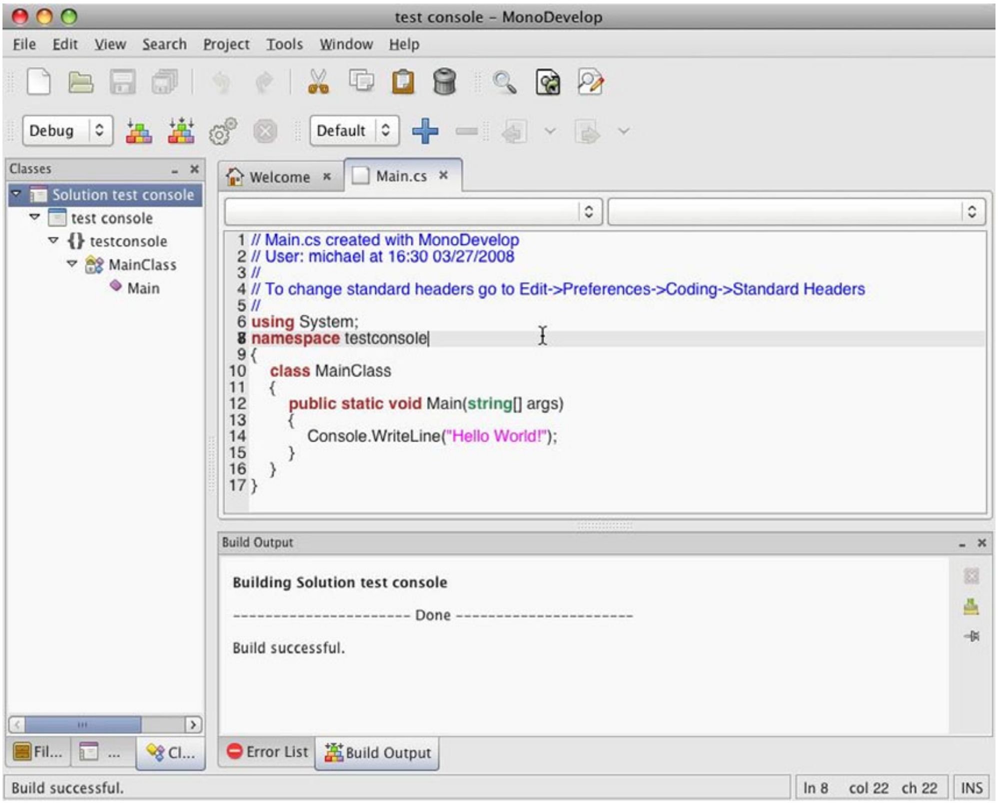


Figure 14.2 MonoDevelop on Mac OS X, editing a C# console project

14.1.2 Python objects from class libraries

Because IronPython is specialized for working with .NET objects, it uses the standard .NET mechanisms. This means that if an object implements `IEnumerable`, then we can iterate over it from IronPython; if it provides `ToString`, then Python will use it to produce its string representation; and so on. Even better than that, we can take advantage of some of the magic that IronPython does on our behalf. .NET methods that expect delegates can receive Python functions without the user (from IronPython) needing to be explicitly aware of the delegate. When we pass in a Python function, IronPython will wrap it with a delegate for us.

Listing 14.1 shows the code for a fairly simple class in C#. `PythonClass` is initialized with an integer and a callback delegate. Iterating over instances yields all the even numbers from the initial value downward, *after* calling the delegate we pass in originally.

Listing 14.1 A C# class for use from IronPython

```
using System;
using System.Collections;
namespace PythonExtension
```

```
{
   public delegate int Callback(int input);
   public class PythonClass : IEnumerable        <— This class implements IEnumerable
   {
      private int _value;
      private Callback _callback;
      public PythonClass(int value, Callback callback)   <— Constructor takes a callback and value
      {
         _value = value;
         _callback = callback;
      }
      public override string ToString()        <— Used for string representation
      {
         return String.Format("PythonClass<{0}>", _value);
      }
      public IEnumerator GetEnumerator()       <— Used for iteration
      {
         for (int i = _value; i > 0; i--)
         {
            if (i % 2 == 0)     <— Only even numbers
            {
               yield return _callback(i);
            }
         }
      }
      public static PythonClass operator +(PythonClass a,
                                  PythonClass b)   <— Operator overloading for addition
      {
         return new PythonClass(a._value + b._value,
                         a._callback);
      }
      public Object this[Object index] {<— The indexer
         get {                                          <— Equivalent of Python __getitem__
            Console.WriteLine("Indexed with {0}", index);
            return index;
         }
         set {                                          <— Equivalent of Python __setitem__
            Console.WriteLine("Index {0} set to {1}", index, value);
         }
      }
   }
}
```

This is slightly odd code; the only good excuse for writing it is if you have profiled your Python code and identified a bottleneck that you can speed up by moving it into C#.

It does illustrate lots of good points for us, though. As well as allowing iteration (through the `IEnumerable.GetEnumerator` method it defines) and working with functions cast to the `Callback` delegate by IronPython for us, it also defines the `operator +` method (operator overloading), which allows us to add `PythonClass` instances together.

The last block of code in the class is the syntax for the indexer, which is the C# equivalent of Python's `__getitem__` and `__setitem__` methods. A .NET class with a public indexer is, of course, indexable from IronPython. The syntax `public Object this[Object index]` (with a nested getter and setter like C# properties) declares a public indexer that takes an object as the index, and the getter returns an object. The setter method inside the indexer declaration receives the value being set as an implicit argument called `value` (it has the same type as the return type of the indexer declaration). In general, Python is my first love, but the C# syntax for properties and indexers is nicer than Python's.

Listing 14.2 shows the same code, but this time written using VB.NET.

Listing 14.2 A VB.NET class for use from IronPython

```
Imports System
Imports System.Collections
Imports System.Collections.Generic
Public Delegate Function Callback(ByVal input As Integer) As Integer

Public Class PythonClass
   Implements IEnumerable                    ' This class implements IEnumerable

   Private _callback As Callback
   Private _value As Integer

   Public Sub New(ByVal value As Integer, ByVal callback As Callback)   ' Constructor takes a callback and value
      Me._value = value
      Me._callback = callback
   End Sub

   Public Overrides Function ToString() As String       ' Used for string representation
      Return String.Format("PythonClass<{0}>", Me._value)
   End Function

   Public Shared Operator +(ByVal a As PythonClass, ByVal b As PythonClass)_   ' Operator overloading for addition
      As PythonClass
      Dim value As Integer
      value = a._value + b._value
      Return New PythonClass(value, a._callback)
   End Operator

   Public Function GetEnumerator() As IEnumerator Implements_
     IEnumerable.GetEnumerator                  ' Used for iteration
      Dim i As Integer
      Dim list As New List(Of Integer)
      For i = Me._value To 0 Step -1
         If (i Mod 2 = 0) Then                  ' Only even numbers
            list.Add(Me._callback(i))
         End If
      Next i
      Return list.GetEnumerator()
   End Function

   Default Public Property Item(ByVal index As Object) As Object   ' The indexer
      Get                                       ' Equivalent of Python __getitem__
         Console.WriteLine("Indexed with {0}", index)
```

```
      Return index
    End Get
    Set(ByVal value As Object)
      Console.WriteLine("Index {0} set to {1}", index, value)
    End Set
  End Property

End Class
```

Equivalent of Python __setitem__

The VB.NET code is semantically almost identical to the C#, as they are very similar languages. We use the default namespace,[5] and the iteration is provided by a list instead of `yield return`. The VB.NET syntax for the indexer is very similar to that of C#. It *has* to have the name `Item` and be marked as the `Default` property, and we also explicitly declare the value argument to the setter.

The VB.NET code is a bit less concise than equivalent Python code. My favorite example of this is the declaration of the iterator method. In Python it is `def __iter__(self):`. In VB.NET it is `Public Function GetEnumerator() As IEnumerator Implements IEnumerable.GetEnumerator`!

Figure 14.3 shows our extension class in use with IronPython (whether compiled from C# or VB.NET). A `PythonClass` instance is initialized with a function (that IronPython neatly turns into a `Callback` delegate for us) and an integer. When we iterate over it, by calling `list` on an instance, we can see that our callback function is called on each number before it is yielded. We can also add `PythonClass` instances, which call the `operator +` method we defined, and we can call `str` on them, which calls our `ToString` method.

We can get lots of standard Python behavior by using normal .NET mechanisms or implementing standard interfaces. To implement a Python mapping type (such as the Python dictionary), you can either use a public indexer or implement the `IDictionary`

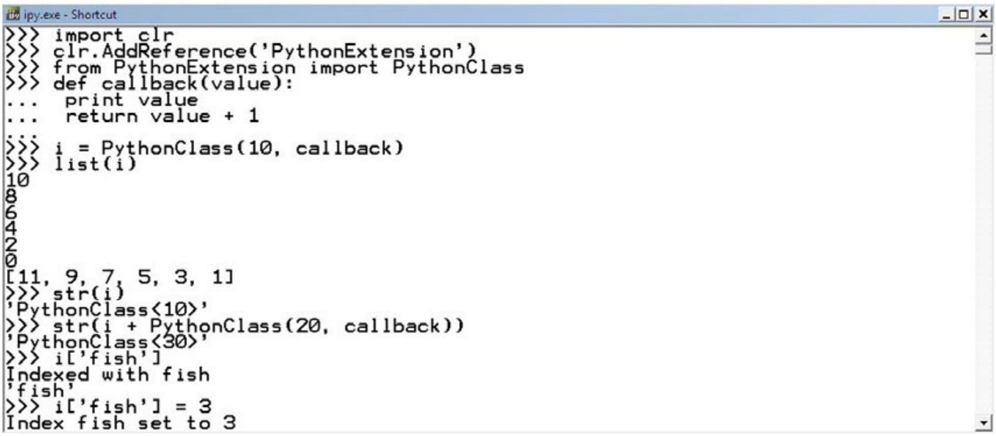


Figure 14.3 `PythonClass` in use from IronPython with a Python callback function

[5] We could have also used the default namespace in C#. All that changes is the way we import the class from IronPython, as you'll see shortly.

interface. To implement a sequence type, you can implement `IList`, and so on. IronPython also defines and uses several interfaces that are special.

For example, although `ToString` is used when `str` is called on an instance of our class, it isn't used for `repr`. You can control the output of `repr` by implementing the IronPython interface `ICodeFormattable`. This is part of IronPython 2, and it means referencing the IronPython assemblies, which is fine if your class is going to be used only from IronPython, but not so good if you are creating a general-purpose class library. If you are happy to tie your objects to IronPython, then the easiest way of working out what interfaces you should use is to browse the IronPython source code. If you look at List.cs from the IronPython 2 source, you can see that the Python list implements the following interfaces: `IMutableSequence`, `IList`, `ICodeFormattable`, `IValueEquality`, and `IList<object>`.

Most of the Python-specific interfaces are defined in Interfaces.cs in the IronPython sources. You'll see that many of the Python interfaces require you to define the Python "magic methods" that you're already familiar with; for example, implementing `ICodeFormattable` means providing a `__repr__` method.

> **Interfaces implemented by Python types**
>
> Knowing which interfaces the basic Python types implement can be really valuable when you're interacting with Python objects from other .NET languages. This is something we look at more closely in the next chapter, but if you can cast an object you pull out of the Python engine to a known type (interface), then you can use those methods from C#/VB.NET.

In fact, we can provide standard Python features on .NET classes *without* having to implement these specific interfaces. This is something we'll cover shortly.

So far we've done nothing that we couldn't also have done from pure Python. In the next section we use .NET attributes to call into unmanaged code, something that we can't do directly with IronPython.

14.1.3 Calling unmanaged code with the P/Invoke attribute

The inability to use .NET attributes is an annoying hole in the IronPython .NET integration. Fortunately it is almost always easy to overcome by writing a small amount of C# and either subclassing or wrapping from Python.

One important attribute is `DllImport`,[6] which is also known as P/Invoke (Platform Invoke), used for calling into unmanaged code.[7] If you need to interact with native

[6] This attribute is documented at http://msdn.microsoft.com/en-us/library/system.runtime.interopservices.dllimportattribute.aspx.

[7] The FePy project does contain an experimental library for doing *dynamic* platform invoke that doesn't require any C#. It needs some work to get it functioning on Windows, though, and means knowing more about topics like C calling conventions than most of us want to have to learn.

code, such as the Win32 APIs on Windows, you can decorate class methods with this attribute and use functions from native dlls.

One of the places where we use this at Resolver Systems is in our test automation framework. We use functions in the User32.dll to interact with Win32 windows. This includes making sure our forms are the foreground window (so that mouse move and mouse button events we send go to our form), closing windows, and getting the title of the topmost window (so that we can confirm that it is our form).

Some of this functionality is available in the `System.Windows.Automation` namespace, but this is new to .NET 3.0, and we need our test framework to be able to run on machines that have only.NET 2.0 installed.

`DllImport` makes it very easy to expose native functions. Listing 14.3 is an example of using Win32 APIs from User32.dll. It exposes sufficient functions for us to be able to get the title of the current topmost window.

Listing 14.3 A thin wrapper that exports functions from User32.dll in C#

```
using System;
using System.Text;
using System.Runtime.InteropServices;

namespace WindowUtils
{
  public class WindowUtils
  {
    [DllImport("user32.dll")]
    public static extern bool IsWindowVisible(IntPtr hWnd);

    [DllImport("user32.dll")]
    public static extern IntPtr GetTopWindow(IntPtr hWnd);

    [DllImport("user32.dll")]
    public static extern IntPtr GetWindow(IntPtr hWnd, uint wCmd);

    [DllImport("user32.dll", SetLastError = true, CharSet = CharSet.Auto)]
    public static extern int GetWindowTextLength(IntPtr hWnd);

    [DllImport("user32.dll", SetLastError = true, CharSet = CharSet.Auto)]
    public static extern int GetWindowText(IntPtr hWnd, [Out] StringBuilder
      lpString, int nMaxCount);

    [DllImport("user32.dll", SetLastError = true, CharSet = CharSet.Auto)]
    public static extern int GetClassName(IntPtr hWnd, [Out] StringBuilder
      lpString, int nMaxCount);
    }
}
```

As you can see, this is a very thin wrapper around the native code we are calling. The native functions are exposed by declaring them as static methods marked as `extern` and with the same name and signature as the native functions. In order to do anything useful with these functions, we're going to write some more code, but now that they're available, we can write that code in Python rather than C#.

Listing 14.4 shows the same code using VB.NET instead of C#. The code, again, is virtually identical. The main difference is that we don't need to declare the out parameters because VB.NET initializes them for us.

Listing 14.4 A thin wrapper that exports functions from User32.dll in VB.NET

```
Imports System
Imports System.Text
Imports System.Runtime.InteropServices

Public Class WindowUtils

   <DllImport("user32.dll")> _
   Public Shared Function IsWindowVisible(ByVal hWnd As IntPtr) As Boolean
   End Function

   <DllImport("user32.dll")> _
   Public Shared Function GetTopWindow(ByVal hWnd As IntPtr) As IntPtr
   End Function

   <DllImport("user32.dll")> _
   Public Shared Function GetWindow(ByVal hWnd As IntPtr, ByVal wCmd As
     UInteger) As IntPtr
   End Function

   <DllImport("user32.dll", SetLastError:=True, CharSet:=CharSet.Auto)> _
   Public Shared Function GetWindowTextLength(ByVal hWnd As IntPtr) As Integer
   End Function

   <DllImport("user32.dll", SetLastError:=True, CharSet:=CharSet.Auto)> _
   Public Shared Function GetWindowText(ByVal hWnd As IntPtr, ByVal lpString As
     StringBuilder, ByVal nMaxCount As Integer) As Integer
   End Function

   <DllImport("user32.dll", SetLastError:=True, CharSet:=CharSet.Auto)> _
   Public Shared Function GetClassName(ByVal hWnd As IntPtr, ByVal lpString As
     StringBuilder, ByVal nMaxCount As Integer) As Integer
   End Function

End Class
```

_ is the VB.NET line continuation symbol

From Python, we can call these functions as static methods on the `WindowUtils` class. They either take or return us a window handle (as an `IntPtr`), and in the case of `GetWindowText` and `GetClassName` they also take a `StringBuilder` to put text into. Because these are C functions, they can't just take an arbitrary string builder, but they also need to know the maximum length of the string they will be populating it with (good old manual memory management at work). In the case of `GetClassName`, we know that the Window class names are all less than 256 characters, so we can simply pass in a string builder initialized with a capacity of 256. Window titles can be an arbitrary length, so we have to make two calls. First we call `GetWindowTextLength`, which tells us the length of the string, and then we can call `GetWindowText`, passing in a string builder with the correct capacity.

There is a little additional complexity in the Python code that uses these functions. The full code is shown in listing 14.5, and the top-level function is `GetTopWindowTitle`.

It gets a handle for the topmost window by calling the aptly named `GetTopWindow`. This turns out to be a surprisingly useless functionality on its own, because the topmost window is inevitably some invisible system window that we aren't interested in at all. We can remedy this by using the `GetWindow` function and the magic `GW_HWNDNEXT` constant that will return us the handle of the *next* window. We simply iterate through all these windows until the `_ExcludeWindow` function finds us one that is both visible and not a system window. Having gotten the handle of the topmost visible and interesting window, we can call `GetWindowText` and return its title.

Listing 14.5 Automation code getting the top window title by calling into native functions

```
import clr
clr.AddReference('WindowUtils')
from WindowUtils import WindowUtils

from System import IntPtr
from System.Text import StringBuilder

GW_HWNDNEXT = 2
def GetTopWindowTitle():
   handle = WindowUtils.GetTopWindow(IntPtr.Zero)
   while handle != IntPtr.Zero:
      if not _ExcludeWindow(handle):          <- Filter uninteresting windows
         break
      handle = WindowUtils.GetWindow(handle, GW_HWNDNEXT)     <- Get the next window handle

   if handle != IntPtr.Zero:
      return GetWindowText(handle)
   return ''

excludes = 'button', 'tooltip', 'sysshadow', 'shell_traywnd'
def _ExcludeWindow(handle):
   if not WindowUtils.IsWindowVisible(handle):     <- Ignore invisible windows
      return True
   class_name = GetWindowClassName(handle)
   for entry in excludes:
      if entry in class_name.lower():     <- Ignore some system windows
         return True
   return False

def GetWindowClassName(hWnd):
   length = 255
   sb = StringBuilder(length + 1)
   WindowUtils.GetClassName(hWnd, sb, sb.Capacity)
   return sb.ToString()

def GetWindowText(hWnd):
   length = WindowUtils.GetWindowTextLength(hWnd)     <- Get the length of the title
   sb = StringBuilder(length + 1)
   WindowUtils.GetWindowText(hWnd, sb, sb.Capacity)     <- Finally get the title
   return sb.ToString()
```

You can see from figure 14.4 that all this actually works!

This technique works fine where we can expose the functionality we need with a thin wrapper and then use it directly in Python. It doesn't work so well where we want

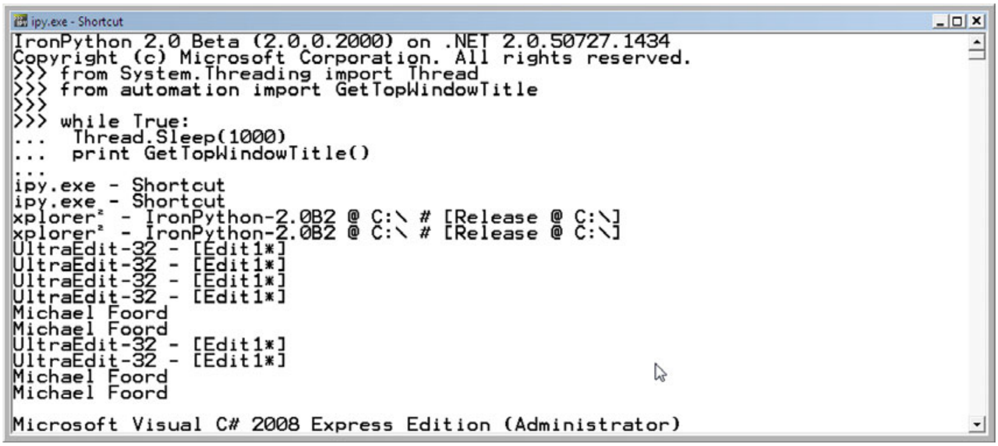


Figure 14.4 Printing the top window title once a second

to apply an attribute to a class, or one of its members, which we want to write in Python. In order to achieve this, we still need to write some statically typed code where we can apply the attributes, but we can then subclass from Python.

14.1.4 Methods with attributes through subclassing

You've seen that we can subclass .NET objects in IronPython, and we can use this to work with attributes. One place this is useful is in Silverlight for communication between IronPython and JavaScript. In the last chapter we looked at several different ways that IronPython and JavaScript can interact, but we didn't cover every possible scenario. In particular, you may want to call into IronPython from JavaScript and have a value returned.

Doing this opens up an interesting possibility. Complex business logic in a client-side application can be written in Python and executed inside Silverlight, where it will run much faster than the equivalent code written in JavaScript. The user interface of the application, or perhaps just part of it, can be written with JavaScript UI libraries, but calling into IronPython to do the heavy lifting. The Silverlight integration could even be optional, acting as an accelerator if present, so long as you are happy to write your code in both JavaScript *and* IronPython!

We expose code from Silverlight to JavaScript by creating instances of types marked with the `ScriptableType` attribute, with methods marked with the `ScriptableMember` attribute, and we have the same old problem of trying to use attributes from IronPython.

We can overcome this problem by writing a very small amount of C# or VB.NET. To do this from Visual Studio we need the professional version, with Silverlight Tools[8] installed. We can then create new class library projects for Silverlight, as you can see in figure 14.5.

8 See http://go.microsoft.com/fwlink/?LinkId=120319.

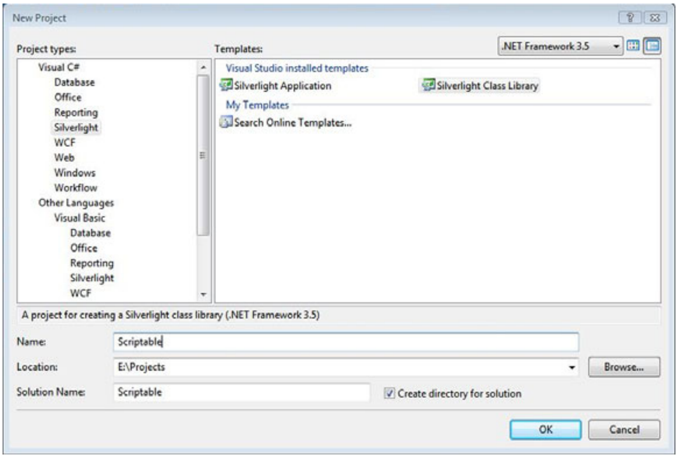


Figure 14.5 Creating a Silverlight class library project with VS 2008 Pro

We can pass only the basic datatypes like strings and numbers between Silverlight and JavaScript. This restriction isn't a problem, though, because we can pass strings to JavaScript. We can construct anything we want in the form of JSON strings, which JavaScript can eval (deserializing with a JavaScript JSON library would be better, of course).

Listing 14.6 shows a C# class with the `ScriptableType` attribute (from `System.Windows.Browser`) and a method with the `ScriptableMember` attribute. `method` takes and returns a string but actually delegates to a method called `real`. Because `real` is a virtual method, we can override it to provide the real implementation in Python.

> **IronPython classes and interfaces**
>
> It is worth noting at this point that we don't need to use this technique for IronPython classes to implement interfaces. An IronPython class can implement a .NET interface by subclassing it and providing the required methods.

Listing 14.6 A stub C# class marked with scriptable attributes

```
using System;
using System.Windows.Browser;

namespace Scriptable
{
    [ScriptableType]
    public class Scriptable
    {
        [ScriptableMember]
        public string method(string value)
        { return this.real(value); }

        public virtual string real(string value)
        { return "override me"; }
    }
}
```

Listing 14.7 is the same class written in VB.NET. Instead of `virtual`, the delegated method `real` is marked as `Overridable`.

Listing 14.7 A stub VB.NET class marked with scriptable attributes

```
Imports System
Imports System.Windows.Browser

<ScriptableType()> _
Public Class Scriptable
   Public Function method(ByVal value As String) As String
      Return real(value)
   End Function

   <ScriptableMember()> _
   Public Overridable Function real(ByVal value As String) As String
      Return "override me"
   End Function
End Class
```

So how do we use this? Well, as with any other assembly, we have to add a reference to it, followed by importing the `Scriptable` class we want to use. We can then subclass `Scriptable` and provide a useful implementation of the `real` method that receives and returns a string when called from JavaScript. The important step is calling `HtmlPage.RegisterScriptableObject`, which exposes objects to the outside world. The first argument to `RegisterScriptableObject` is the name our object will be exposed with, and the second is an instance of a scriptable class. Listing 14.8 shows the IronPython code that makes use of our `Scriptable` class.

Listing 14.8 Exposing a scriptable class and method to JavaScript

```
import clr
clr.AddReference("Scriptable")
from Scriptable import Scriptable

from System.Windows.Browser import HtmlPage

class ScriptableClass(Scriptable):            <-- Subclass the class with scriptable attributes
   def real(self, string):                    <-- Override the virtual real method
      ...
      return new_string                       <-- Provide a useful implementation

instance = ScriptableClass()

HtmlPage.RegisterScriptableObject("scriptable", instance)     <-- Register the scriptable instance
```

Having registered the object from inside Silverlight, we can now access it from the outside. The following snippet of JavaScript fetches the Silverlight control by id (one good reason to give the control itself an id when you embed it in the web page) and then calls the exposed method:

```
control = document.getElementById('SilverlightPlugin');
result = control.Content.scriptable.method(value);
```

Note that the JavaScript calls `method`, which has the `ScriptableMember` attribute applied, rather than `real`, which actually does the work.

This is all fine and dandy if you *have* Visual Studio Professional, but it's something of a problem if you don't. Although Visual Studio and Silverlight Tools provide a convenient way of compiling class libraries for Silverlight, this isn't the only way.

COMPILING ASSEMBLIES WITHOUT VISUAL STUDIO

Having to have Visual Studio 2008 just to compile a few lines of C# is a nuisance. Fortunately, a C# compiler is a standard part of a normal .NET install, and we can use this directly.

The C# compiler is called csc.exe. We can pass in command-line arguments that tell it not to reference the standard .NET assemblies but use the Silverlight ones instead.

Listing 14.9 is a batch file[9] that compiles any C# files in the current directory (*.cs) into an assembly specified by the `/out` argument. The `/nostdlib+ /noconfig` arguments tell csc not to use the standard framework assemblies, and the `/r` arguments add explicit references to the Silverlight assemblies we are using.[10]

Listing 14.9 A batch file for compiling .cs files into assemblies for Silverlight

```
set sl=C:\Program Files\Microsoft Silverlight\2.0.31005.0
set csc=C:\Windows\Microsoft.NET\Framework\v2.0.50727\csc.exe

%csc% /t:library /out:Scriptable.dll
   /nostdlib+ /noconfig
   /r:"%sl%\mscorlib.dll" r:"%sl%\System.dll"
   /r:"%sl%\System.Core.dll"
   /r:"%sl%\System.Net.dll"
   /r:"%sl%\System.Windows.Browser.dll"
   *.cs

pause
```

Annotations: Location of the Silverlight assemblies (line 1); Path to csc.exe, the C# compiler (line 2); All C# files in this directory (`*.cs`)

This is very simple to automate as part of your build process, and it minimizes the difficulty of maintaining the small parts of a project that can't be kept in pure Python.

We've been looking at creating class libraries for use from IronPython. C# and VB.NET are, like Python, imperative object-oriented programming languages. If you know one, the core concepts are similar enough that the others are easy to learn. Despite the similarities, there are also many differences, and there are times when one language is more appropriate than another. With the help of the Dynamic Language Runtime, the .NET framework is a great environment to engage in "polyglot programming."[11] In this section we've shown several good reasons to write code in C#/VB.NET for use from IronPython, whether for performance reasons or to use features of the CLR that we can't access directly from IronPython. In fact, using .NET attributes, we can even use C# to access code written in C.

---

9 The lines that start with `%csc%` should be on one line; I've split the code into multiple lines here for readability.

10 The exact location (on disk) of the Silverlight assemblies will depend on the version you have installed. The directory path shown in listing 14.9 is the location for Silverlight 2.

11 One of the first references to polyglot programming on the .NET framework dates from 2002, the same year as the release of version 1.0 of the .NET framework. See http://www.ddj.com/architect/184414854.

When we write .NET libraries for IronPython, they are generally usable with no special effort, but there are things that that we can do to make these objects feel more "Pythonic."

14.2 Creating dynamic (and Pythonic) objects from C#/VB.NET

When you create a normal Python class, you can add whatever attributes you want at runtime, as the following interactive session illustrates:

```
>>> class SomeClass(object): pass
>>> instance = SomeClass()
>>> instance.x = 3
>>> instance.x
3
```

We can also define the magic methods `__setattr__`, `__getattr__`, and `__delattr__` to customize what happens when arbitrary attributes are set, fetched, or deleted. We can't do that with the classes we've been writing in C# and VB.NET; attempting to do so will throw an attribute error. To be fair, the same thing will happen with Python classes implemented in C extension modules (or Python classes that define `__slots__`), but CPython does provide ways for extension classes to provide dynamic attribute access—and so does IronPython.

In this section we look at several special methods that you can define on .NET classes to permit, and customize, dynamic attribute access. Implementing these provides a nice, Python-friendly API, while still not being *dependent* on IronPython and allowing classes to be used from other languages. As well as using the dynamic attribute access methods, you can use argument attributes to make your method signatures feel more native to the Python programmer.

14.2.1 Providing dynamic attribute access

Dynamic attribute access can be used to create a natural-feeling API, like accessing DOM elements by name on the `Document` class in Silverlight.

There are actually five methods that a .NET class can implement to provide dynamic attribute access.[12] These methods are

- *`GetCustomMember`*—Runs before normal .NET lookups
- *`GetBoundMember`*—Runs after normal .NET lookups
- *`SetMember`*—Runs before normal .NET member assignment
- *`SetMemberAfter`*—Runs after normal .NET member assignment
- *`DeleteMember`*—Runs before normal .NET operator access (there's no .NET equivalent)

These methods aren't part of an interface, but instead you mark them with the `SpecialName` attribute.[13] This is a standard .NET attribute (living in the `System.`

[12] Special thanks to Srivatsn Narayanan, from the IronPython team, for his help with this section.
[13] See http://msdn.microfost.com/en-us/library.system.runtime.compilerservices.specialnameattribute.aspx.

Runtime.CompilerServices namespace), and so using it doesn't require us to reference the IronPython assemblies.

Because you almost certainly don't want dynamic attribute access to interfere with the normal use of your class (access to members that you have defined normally), you will probably want to implement GetBoundMember/SetMemberAfter. These are run *after* normal .NET lookup has been tried and are called only if the name doesn't exist.

You can use GetCustomMember and SetMember (called before normal .NET lookups) to override normal member lookup. With GetCustomMember you can signal to IronPython that it should proceed with normal lookup by returning OperationFailed.Value.[14] OperationFailed *is* defined inside the IronPython assemblies, and so you will need to reference them to use it. SetMember can signal that normal member lookup should proceed by returning a bool; True signifies that no further lookups are required and False signifies that normal lookup should continue. If SetMember returns anything other than a bool, then no further lookups will be attempted.

Listing 14.10 is a class that uses GetBoundMember, SetMemberAfter, and DeleteMember to allow you to fetch, set, and delete members.

Listing 14.10 A C# class that allows dynamic attribute access from IronPython

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Runtime.CompilerServices;

namespace DynamicObject
{
   public class DynamicObject
   {
     private Dictionary<string, object> _dict = new Dictionary<string, object>();

      public Dictionary<string, object> container
      {
         get { return _dict; }          <- Fetch the underlying attribute dictionary
      }

      [SpecialName]
      public object GetBoundMember(string name)       <- Attribute fetch method
      {
         if (!_dict.ContainsKey(name))        <- Raise an AttributeError for missing attributes
         {
            string msg = String.Format("'DynamicObject' has no attribute '{0}'",
               name);
            throw new System.MissingMemberException(msg);
         }
         return _dict[name];        <- Return the requested attribute
      }
```

[14] GetBoundMember can also return OperationFailed.Value, which will signal to IronPython to raise an AttributeError.

```
        [SpecialName]
        public void SetMemberAfter(string name, object o)
        {
            _dict.Add(name, o);
        }

        [SpecialName]
        public void DeleteMember(string name)
        {
            if (!_dict.ContainsKey(name))
            {
                string msg = String.Format("'DynamicObject' has no attribute '{0}'",
                    name);
                throw new System.MissingMemberException(msg);
            }
            _dict.Remove(name);
        }
    }
}
```

Method for setting attributes

Method for deleting attributes

The class DynamicObject stores attributes in a dictionary (just as they would be stored in the __dict__ dictionary on a Python object), which is exposed via the container property. The three special methods for fetching, setting, and deleting attributes manipulate the underlying dictionary. Both DeleteMember and GetBoundMember check to see if an attribute exists before attempting to fetch it or delete it from the dictionary. If the attribute is not found, then it raises a MissingMemberException (which becomes an AttributeError in IronPython) rather than a KeyNotFoundException. Returning OperationFailed.Value here would have the same effect but would require us to reference the IronPython assemblies.

The following interactive session shows our DynamicObject class in action:

```
>>> import clr
>>> clr.AddReference('DynamicObject')
>>> from DynamicObject import DynamicObject
>>> d = DynamicObject()
>>> d.attribute = 'an attribute'
>>> print d.attribute
an attribute
>>> del d.attribute
>>> d.attribute
Traceback (most recent call last):
AttributeError: 'DynamicObject' has no attribute 'attribute'
```

Because the attributes are stored in a publicly exposed dictionary, a C# application could use DynamicObject by directly working with container (an alternative would be to use an indexer and allow both indexing and attribute access). This way we have created an API that is usable from all .NET languages but uses the dynamic features of Python when used from IronPython.

Listing 14.11 is the same class implemented in VB.NET.

Listing 14.11 A VB.NET class that allows dynamic attribute access from IronPython

```
Imports System
Imports System.Collections.Generic
Imports System.Text
Imports System.Runtime.CompilerServices

Public Class DynamicObject

   Private _error As String = "'DynamicObject' has no attribute '{0}'"
   Private _instance_dict As New Dictionary(Of String, Object)
   Public ReadOnly Property container() As Dictionary(Of String, Object)   ◁ Fetch the underlying attribute dictionary
      Get
         Return _dict
      End Get
   End Property

   <SpecialName()> _
   Public Function GetBoundMember(ByVal name As String) As Object   ◁ Attribute fetch method
      If Not _instance_dict.ContainsKey(name) Then   ◁ Raise an AttributeError for missing attributes
         Dim msg As String = String.Format(_error, name)
         Throw New System.MissingMemberException(msg)
      End If
      Return _instance_dict.Item(name)   ◁ Return the requested attribute
   End Function

   <SpecialName()> _
   Public Sub SetMemberAfter(ByVal name As String, ByVal value As Object)   ◁ Method for setting attributes
      _instance_dict.Add(name, value)
   End Sub

   <SpecialName()> _
   Public Sub DeleteMember(ByVal name As String)   ◁ Method for deleting attributes
      If Not _instance_dict.ContainsKey(name) Then
         Dim msg As String = String.Format(_error, name)
         Throw New System.MissingMemberException(msg)
      End If
      _instance_dict.Remove(name)
   End Sub

End Class
```

There is a sixth method, related to dynamic attribute access, that we can also implement. In Python we determine the members available on an object by calling `dir` on it. This doesn't play well with objects that dynamically create members, like the code we have just written (pure Python code has the same problem with objects that use `__getattr__` for attribute access). Calling `dir` on a `DynamicObject` instance shows only the public members we have explicitly defined (plus default members that exist on all objects) and not the ones that were created by `SetMemberAfter`. The sixth method is `GetMemberNames`, and it will be used when `dir` is called on an object that implements it. Like the other methods, it must be marked with the `SpecialName` attribute, and it should return an `IEnumerable` (of string) that lists all of the member names on the object.

These special methods mirror the Python magic methods `__getattr__` and `__setattr__`.[15] There are plenty of other Python protocols that you might want to support; let's see how you do that.

14.2.2 Python magic methods

The intention is that you should be able to provide any Python magic method on a C# or VB.NET class simply by implementing it.

> **Protocol methods on .NET classes with IronPython 2**
>
> I (Michael) say *intention* because during the early IronPython 2 Betas, it was still necessary to implement interfaces for some of the protocol methods, such as `ICodeFormattable` to support `repr`. By IronPython 2, if there is any protocol method (other than the exceptions) that can't be implemented directly, it will be a bug.

So if we could implement `__getattr__` directly, why did we first look at `GetBoundMember`, `SetMember`, and friends? Well, there are a few exceptions—and that includes the get/set/delete member customizers that we covered in the last section. On top of this they are DLR methods rather than being Python specific. Classes that implement these methods will work with any language that runs on the Dynamic Language Runtime.

There is one more exception, too, the `__call__` method, which you can implement with a `Call` method marked with the `SpecialName` attribute. The following snippet of C# shows the `Call` method definition:

```
[SpecialName]
object Call(string someArg, int someOtherArg)
```

In general, if there's a .NET interface or operator method that maps onto the Python methods, you should use it. You've already seen how .NET indexing maps to the Python `__getitem__` and `__setitem__` methods. Instead of `__enter__`/`__exit__`, you can implement `IDisposable` and so on. This mapping of .NET interfaces and methods into Python methods is done in the source file TypeInfo.cs.

In implementing these methods we're explicitly providing features to create objects that are easy and intuitive to use from Python. What *intuitive* means in a programming context is "familiar," and this means using the techniques and patterns that are common to Python. This can be a hard goal when writing in a language that has its own idioms. There is more to the IronPython .NET integration that we can take advantage of to achieve it.

14.2.3 APIs with keyword and multiple arguments

One way to create a flexible and easy-to-use API in Python is through the use of keyword arguments. Keyword arguments serve two purposes in a single mechanism. They

[15] `GetMemberNames` mirrors the `__dir__` magic method added in Python 2.6.

make individual arguments to a function or method optional (supplying a default value if the argument is not passed) and make argument passing more explicit by allowing them to be passed in by name. Python also allows you to collect all arguments passed in using the `*args` notation.

Constructors and keyword arguments

While we're talking about keyword arguments, don't forget that IronPython already gives special treatment to keyword arguments passed into constructors. Additional arguments passed to a constructor by keyword will be used to set properties on the object after it has been created.

```
i = SomeClass(arg1, X=value)
```

is the equivalent of

```
i = SomeClass(arg1)
i.X = value
```

.NET supports the latter, through the `params` keyword in C# and `ParamArray` in VB.NET, but it doesn't directly support the concept of keyword arguments. Both VB.NET and C# have *some* concept of optional arguments, though, and IronPython again does magic on our behalf to make them compatible with Python keyword arguments. This means that we can write .NET classes in C# and VB.NET that accept keyword arguments when used from IronPython.

In C# we do this by marking parameters with the `DefaultParameterValue` attribute[16] from the `System.Runtime.InteropServices` namespace. This allows us to specify a value that will be used if the argument is not passed at call time. Interestingly, we *can't* omit the argument when calling from C#. Although C# allows us to create methods with parameters marked in this way, the language itself doesn't support optional arguments.

This attribute works fine in most cases. One situation it doesn't work in is where the parameter is of a nullable type. A nullable type is a special version of a value type (like an integer or a Boolean, which can't normally be represented by a null) that allows null as a valid value. Under the hood .NET boxes the value, so there are performance implications, but they can be extremely useful where you need null to represent a special value. In C# you declare a variable to be of a nullable type by adding a question mark to the type declaration. A nullable integer is declared with the type declaration `int?`. Unfortunately we can't use the `DefaultParameterValue` attribute with nullable types, but we can use the `Optional` attribute[17] instead. This doesn't allow us to specify a default value, but you can always check for null inside the body of your method and supply a default yourself.

---

16 See http://msdn.microsoft.com/en-us/library/system.runtime.interopservices.defaultparametervalueattribute.aspx.

17 See http://msdn.microsoft.com/en-us/library/system.runtime.interopservices.optionalattribute.aspx.

Listing 14.12 shows a C# class with three static methods. The first takes strings as arguments but provides default values for the last two. The second method takes nullable integers as its last two arguments. It prints all three arguments it is called with, but if the second argument is null (omitted), then it is replaced with a default value inside the body of the method.[18] The third method allows you to call it with as many arguments as you want, using the `params` keyword, and prints out how many arguments it is called with.

Listing 14.12 Methods with multiple arguments and default arguments from C#

```
using System;
using System.Runtime.InteropServices;

namespace DefaultArguments
{
   public class Example
   {
      public static void DefaultValues(string string1,
       [DefaultParameterValue("default")] string string2,
       [DefaultParameterValue("another")] string string3)
      {
         Console.WriteLine("1st argument = " + string1);
         Console.WriteLine("2nd argument = " + string2);
         Console.WriteLine("3rd argument = " + string3);
      }

      public static void OptionalIntegers(int value1,
         [Optional] int? value2, [Optional] int? value3)
      {
         if (!value2.HasValue)
         {
            value2 = 100;
         }
         Console.WriteLine("1st argument = {0}", value1);
         Console.WriteLine("2nd argument = {0}", value2);
         Console.WriteLine("3rd argument = {0}", value3);
      }

      public static void MultipleArguments(params object[] args)
      {
         int len = args.Length;
         Console.WriteLine("You passed in {0} args", len);
         for (int i = 0; i < len; i++)
         {
           object v = args[i];
           Console.WriteLine("Argument {0} is {1}", i, v);
         }
      }
   }
}
```

Provide a default value of "default"

Optional arguments for nullable integers

Provide a default for null values

Collects arguments as an object array

Print all the arguments

[18] The example tests the nullable using `!value2.HasValue`. A more concise alternative is the C# null-coalescing operator: `value2 = value2 ?? 100`.

When this is compiled, we can call these methods from IronPython, passing in arguments by keyword. Of course in Python, and similarly in IronPython, we can *always* pass in arguments by name. The important difference is that with arguments marked with these attributes we can omit altogether the ones that have default values.

You can see how they behave in practice from the output from the following interactive session. Note particularly that when calling `DefaultValues` and `OptionalIntegers`, we can omit the second argument and pass in the third by keyword.

```
>>> import clr
>>> clr.AddReference('DefaultArguments')
>>> from DefaultArguments import Example
>>> Example.DefaultValues('first', string3='third')
First argument = first
Second argument = default
Third argument = third
>>> Example.OptionalIntegers(10, value3=2)
First argument = 10
Second argument = 100
Third argument = 2
>>> Example.MultipleArguments(1, 'two', object(), 4)
You passed in 4 arguments
Argument 0 is 1
Argument 1 is two
Argument 2 is System.Object
Argument 3 is 4
```

VB.NET does have language support for optional arguments. To a certain extent this makes things easier; we can just declare arguments as optional and supply a default value. These behave like arguments marked with the `DefaultParameterValue` attribute, which means that we can't use them with nullable types.

Optional arguments in VB.NET

If you read about optional arguments in VB.NET, you will find that they are usually not recommended. The main reason for this is that optional arguments aren't usable if you consume the API from C#. In addition, the optional values are stored as metadata in the IL bytecode and actually compiled into the methods' *callers* (meaning changes to default values require their callers to be recompiled as well). Because IronPython resolves them dynamically, it doesn't suffer from these problems.

In fact, the use of optional arguments is looked down on (sorry, I mean "is not recommended") in VB.NET. The recommended technique is to use multiple overloads. You can supply one overload that takes all the arguments and alternative overloads that take fewer arguments but fill in the missing ones with default values. In this case IronPython allows us to pass in arguments by name and works out for us which overload to call. To make more than one argument optional, you will have to implement a different overload for every possible combination of arguments, and each overload

must have a different type signature. Unfortunately, this makes having several optional arguments of the same type impossible just through overloads.

Listing 14.13 shows what we can do with VB.NET. The `Example` class here has `DefaultValues` and `MultipleArguments` methods that do the same as their C# equivalents.

Listing 14.13 Optional and multiple arguments from VB.NET

```
Public Class Example
   Public Shared Sub DefaultValues(ByVal string1 As String, _
           Optional ByVal string2 As String = "default", _
           Optional ByVal string3 As String = "another")
      Console.WriteLine("First argument = " + string1)
      Console.WriteLine("Second argument = " + string2)
      Console.WriteLine("Third argument = " + string3)
   End Sub

   Public Shared Sub MultipleArguments( _
           ByVal ParamArray args() As Object)
      Dim len As Integer = args.Count
      Console.WriteLine("You passed in {0} arg", len)
      For i As Integer = 0 To UBound(args, 1)
         Dim arg As Object = args(i)
         Console.WriteLine("Argument {0} is {1}", i, arg)
      Next
   End Sub
End Class
```

Arguments declared as optional

Taking multiple arguments with ParamArray

Print all the arguments

Designing an API is one of the most important tasks in programming. It influences the structure and usability of your application or libraries. This is why I am a fan of test-driven development; it makes you think about the usability of your API *before* you think about the implementation.

We've been looking at ways to give an API written in C# or VB.NET that elusive Pythonic quality when used from IronPython. Of course, the easiest way to achieve this is to write your code in Python in the first place, but it may not always be possible. We have already shown how using attributes from IronPython requires the writing of at least *some* code in a more traditional .NET language, whether it be a stub class or a thin wrapper around native functions. In particular, writing stub classes can feel like mechanical work, ripe for automation. In fact, we can automate the compiling of these classes from text (which, after all, is what the compiler does), which means we could automate the generation of the stubs at runtime!

In the next section we use some of the APIs available in .NET that allow us to dynamically compile (and then use) code at runtime.

14.3 Compiling and using assemblies at runtime

The code we wrote earlier to wrap native functions for `WindowUtils` was brief, but having to maintain a separate Visual Studio project and recompile and then copy assemblies across every time we changed the code could be annoying. We can avoid this, and actually eliminate the need to even save binary assemblies at all, by compiling straight from source code into memory!

.NET provides access to the compiler infrastructure, through the `System.CodeDom.Compiler` namespace[19] in conjunction with an appropriate code provider. Accessing this programmatically from IronPython is straightforward once you know the magic invocations.

Listing 14.14 is a `Generate` function that uses either a `CSharpCodeProvider` or a `VBCodeProvider`, configured with `CompilerParameters` initialized from the options chosen in the function call signature. It takes the source code and an assembly name, and depending on the options you pass in, it will return either the compiled assembly or the path to the assembly saved on disk.

Listing 14.14 A function to compile, and save or return, assemblies from source code

```
from System.Environment import CurrentDirectory
from System.IO import Path, Directory

from System.CodeDom import Compiler
from Microsoft.CSharp import CSharpCodeProvider
from Microsoft.VisualBasic import VBCodeProvider

def Generate(code, name, references=None, outputDir=None,
          inMemory=False, csharp=True):
   params = Compiler.CompilerParameters()

   if not inMemory:                              # Configure the path to save assembly
      if outputDir is None:
         outputDir = Directory.GetCurrentDirectory()
      asmPath = Path.Combine(outputDir, name + '.dll')
      params.OutputAssembly = asmPath
      params.GenerateInMemory = False
   else:
      params.GenerateInMemory = True

   params.TreatWarningsAsErrors = False
   params.GenerateExecutable = False
   params.CompilerOptions = "/optimize"

   for reference in references or []:            # Add references to assemblies
      params.ReferencedAssemblies.Add(reference)

   if csharp:                                    # Get the appropriate code provider
      provider = CSharpCodeProvider()
   else:
      provider = VBCodeProvider()
   compile = provider.CompileAssemblyFromSource(params, code)   # This is the important bit

   if compile.Errors.HasErrors:                  # Raise an exception with any errors
      errors = list(compile.Errors.List)
      raise Exception("Compile error: %r" % errors)

   if inMemory:
      return compile.CompiledAssembly
   return compile.PathToAssembly
```

The parameters for the `Generate` function are

[19] See http://msdn.microsoft.com/en-us/library/system.codedom.compiler.aspx.

- *code*—This is the source code we are compiling, as a string.
- *name*—The name of the assembly to generate.
- *references*—A list of strings with additional assemblies that your assembly needs to reference. If these assemblies aren't a standard part of the .NET framework, then they will need to be absolute paths to the assemblies on the filesystem.
- *outputDir*—This is an optional argument for a directory to save the assembly to. If you *don't* supply a path here, but `inMemory` is `False`, then the assembly will be saved in the current directory.
- *inMemory*—If this is `True`, then the assembly will be generated in memory, and `Generate` will return the assembly object. If this is `False`, then `Generate` will return the path to the saved assembly instead.
- *csharp*—If this is `True`, then the source code should be C#. If it is `False`, then the source code should be VB.NET.

If we call `Generate` with `inMemory=True` and no `outputDir` parameter, then instead of saving the generated assembly to disk, it compiles into memory and returns us an assembly object. From there we can either access the namespaces contained in the class as attributes on the assembly, or we can take advantage of the fact that `clr.AddReference` allows us to add a reference to an assembly object. After adding a reference to the assembly, we can import from it normally.

Going back to our earlier example of wrapping native functions in the `WindowUtils` class, instead of keeping this as a separate Visual Studio project we can generate the assembly from the source code as a string. We can do that by changing the start of automation.py to the following:

```
import clr
assembly = Generate(source, 'WindowUtils', inMemory=True)
clr.AddReference(assembly)
from WindowUtils import WindowUtils
```

From there on, automation can use `WindowUtils` in exactly the same way as if the assembly had been loaded from disk. If we need to modify the source code, we can simply rerun our script, and although we still have a compile phase, it is done dynamically at runtime rather than as a separate step before we can run our code.

Instead of adding a reference and then importing, we could also do this:

```
assembly = Generate(source, 'WindowUtils', inMemory=True)
WindowUtils = assembly.WindowUtils.WindowUtils
```

Of course, C# can do all of this; in fact IronPython is built on a similar technique but using the `Reflection.Emit` API to generate code directly as IL bytecode. The difference is that with IronPython we can dynamically reference and use assemblies at runtime, whereas C# code must have access to all the classes it uses at compile time—making code generation like this much less useful.

Dynamically compiling at runtime opens up all sorts of interesting possibilities. Consider, for example, the subject of building stub classes, like the one we used for

working with Silverlight attributes. We can now create assemblies from text, and IronPython is an excellent language for manipulating and generating text. We could use introspection on Python classes to autogenerate the stubs for us, decorated with attributes as required. This is left as an exercise for the reader![20]

14.4 Summary

For complete integration with the .NET framework, it is sometimes *necessary* to use another language. Fortunately, with IronPython this is very easy, and not only is it much easier than with our old friend CPython, but these languages are much more pleasant to work with than C.

Even when working with C, CPython doesn't have the advantage over IronPython. CPython has an FFI[21] called `ctypes`,[22] and the .NET equivalent is the Platform Invoke attribute. This requires at least some C# or VB.NET, and if writing this code feels painfully like writing boilerplate, then you can always generate the code automatically and compile at runtime.

We're not generally fans of code generation[23] as a form of metaprogramming; however, in the case of IronPython it could be an ideal way of taking advantage of the benefits of static typing. Translating algorithms or interface code into assemblies, compiled from C# generated at runtime, is a powerful concept.

This chapter has been about writing C#/VB.NET in order to extend IronPython, writing class libraries that we use from inside it. The other side of the coin, and one of the major use cases for IronPython, is to embed the Python engine into a .NET application and interact with it from the outside. This is also easy, but there are lots of different ways of doing this—and you'll be learning about them in the next chapter.

---

[20] Although Curt Hagenlocher has started a project that uses a very similar technique to generate proxy classes, allowing you to use IronPython classes from C#. See http://www.codeplex.com/coils.

[21] The Foreign Function Interface is for calling into native code.

[22] See http://docs.python.org/lib/module-ctypes.html.

[23] Ironically, Resolver One is based on the principle of generating Python code from spreadsheet formulas.

15 Embedding the IronPython engine

This chapter covers

- Creating a custom executable
- The IronPython engine
- The DLR hosting API
- Handling Python exceptions
- Interacting with dynamic objects

Embedding the IronPython engine is one of the major reasons for wanting to use IronPython. It provides a ready-made scripting language engine for embedding into applications. You can use it to provide user scripting and plugins or integrate Python code into systems written in other languages. You could even use it to provide user scripting for Silverlight applications.

Embedding the IronPython engine means turning the world, as we have seen it so far, inside out. Instead of using .NET objects from inside IronPython, we host the language engine inside C# or VB.NET—and interact with Python objects from these languages. This means that we make objects available for the Python code to work with and then work with objects created from Python back on “the other side.”

Although we work specifically with IronPython, much of what we look at is relevant to the other DLR languages as well.

There are many different ways of embedding IronPython, corresponding to the myriad of use cases to which you might apply it. In this chapter we take a leisurely stroll through some of the more common hosting scenarios. Not only will this cover many of the straightforward things you might want to achieve, but it will equip you to experiment with more esoteric uses to which you might bend IronPython.

IronPython is an enormously powerful tool for use in .NET applications. As well as providing ways to extend and script applications, it also allows you to create custom DSLs[1] where business rules can be stored as text without the application having to be recompiled. We'll be covering all of these use cases but starting with one of the simpler use cases: creating an executable that launches a Python application.

15.1 Creating a custom executable

One of the most basic ways of embedding IronPython is to create an executable file that executes a Python file. This allows you to distribute a Python application, where the main entry point is a standard .exe file. Because this uses only a subset of the API, it makes a great first example. Before we tackle the whole code for creating a custom executable, let's look at the major components in the hosting API, starting with the IronPython engine itself.

> **The IronPython hosting API**
>
> In this chapter we work with the IronPython and DLR hosting API, which are part of IronPython 2. All the examples in this chapter use the API from the IronPython 2 Final Release. If there are any changes for subsequent releases, we will post them to the *IronPython in Action* website.[2]

15.1.1 The IronPython engine

To use IronPython we need to include all five assemblies that come with it:

- Microsoft.Scripting.dll
- Microsoft.Scripting.Core.dll
- IronPython.dll
- IronPython.Modules.dll
- Microsoft.Scripting.ExtensionAttribute.dll

The first two assemblies comprise the Dynamic Language Runtime. The next two are specific to IronPython. IronPython.dll implements the Python language syntax and

---

1 Domain Specific Languages—These are "little languages" that encapsulate rules for a specific problem domain.

2 See http://www.ironpythoninaction.com/.

semantics. IronPython.Modules.dll provides a C# implementation of some of the built-in modules that in CPython are written in C.

The last assembly is a bit special; it is there so that .NET projects using IronPython work with both .NET 2.0 and .NET 3.0. The DLR uses extension methods, both because it shares an AST format with LINQ (expression trees) and to provide convenience methods on several parts of the hosting API. The extension methods are decorated with the `ExtensionAttribute` attribute, which is part of .NET 3.0. All the `Microsoft.Scripting.ExtensionAttribute` assembly does is provide this attribute so that IronPython can be compiled under .NET 2.0. The important thing is that it must be included in your projects but not directly referenced. If your project is a .NET 3.0 project, then it will use the standard `ExtensionAttribute`, and if it is a .NET 2.0 project, it will use the one provided by the DLR.

The main entry point for applications that embed IronPython is the `Python` class in the `IronPython.Hosting` namespace. It has convenience methods for creating `ScriptEngine` and `ScriptRuntimes` preconfigured for working with IronPython.[3]

Once we have set up a project that references these assemblies, we can extract a Python language engine from their grasp with the following snippet of C#:

```
using IronPython.Hosting;
ScriptEngine engine = Python.CreateEngine();
```

The VB.NET equivalent is

```
Imports IronPython.Hosting
Dim engine As ScriptEngine = Python.CreateEngine()
```

Examples in C# or VB.NET

This chapter illustrates the IronPython embedding API by working through four different examples. These examples are all available in the chapter 15 folder of the downloadable sources, as Visual Studio projects in *both* C# and VB.NET. Because of space restrictions we only show code examples in either C# or VB.NET, except where there are notable differences between them.

This code creates a single engine. The Dynamic Language Runtime supports the creation of multiple runtimes within a single application. This can be extremely useful for creating execution contexts that are isolated from each other,[4] a feature that Resolver One uses to have multiple open but separate spreadsheets.

In the examples that follow we will cover the most common ways of working with the IronPython hosting API, but you often have a choice of several different routes to achieve the same end. For more complete documentation on *all* the possibilities, you

[3] The IronRuby project has a similar entry point with easy access to engines and runtimes preconfigured for working with IronRuby.

[4] There is also support for creating separate runtimes in different AppDomains for further isolation.

should refer to the source code (this is open source after all) or the "DLR Hosting Spec" document.[5]

Once we have created an engine, we are ready to execute code, either from a string or directly from a file. At this stage our goal is to create an executable that launches a Python application, so we want to run a file.

15.1.2 Executing a Python file

To execute code we must introduce two more components from the hosting API, the `ScriptSource` and the `ScriptScope`. The `ScriptSource` can be created from the engine, either from a source file or from code in a string. A `ScriptScope` represents a Python namespace (a scope) and is also created from the engine.

C# snippets to do this are as follow. First, for executing code from a string:

```
ScriptSource source;
source = engine.CreateScriptSourceFromString(sourceCode,
   SourceCodeKind.Statements);
ScriptScope scope = engine.CreateScope();
source.Execute(scope);
```

To execute Python code in a file:

```
ScriptSource source;
source = engine.CreateScriptSourceFromFile(programPath);
ScriptScope scope = engine.CreateScope();
source.Execute(scope);
```

> **Exploring the IronPython hosting API**
>
> As with the other assemblies we have used in this book, the easiest way of exploring the IronPython hosting API is with the interactive interpreter. From ipy.exe you can add references to the DLR and IronPython assemblies and experiment interactively with these classes.

Figure 15.1 shows a diagram of the DLR hosting API that we've used to get to this point.

This isn't quite enough, however. In order to *properly* execute a Python program, we should do a bit more work and set up things like `sys.path` (the import path) and provide any command-line arguments that were passed to the program.

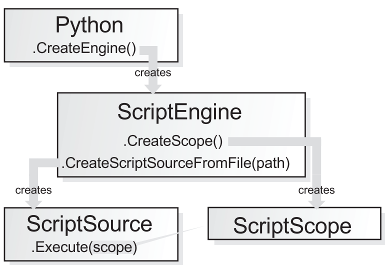


Figure 15.1 The major DLR hosting API components used to execute a Python file

SETTING THE COMMAND-LINE ARGUMENTS

Command-line arguments in a .NET application are passed into the `Main` method of our application as a string array. Python programs

[5] This is available from the DLR project page on CodePlex: http://www.codeplex.com/dlr.

expect sys.argv to be a list of strings, where the first entry is the program filename and the entries following are the command-line arguments.

The Python class provides easy access to the sys module for a Python engine with the GetSysModule method. This method takes the engine as an argument and returns a ScriptScope. Listing 15.1 extracts the sys module and sets the command-line arguments.

Listing 15.1 Setting the command-line arguments for Python code from C#

```
using IronPython.Runtime;

List argList = new List();
argList.extend(args);
ScriptScope sys = Python.GetSysModule(engine);
sys.SetVariable("argv", argList);
```

The value of sys.argv needs to be a Python list. Naturally the Python list type in IronPython is implemented in C#, so we can use it from our embedding code—including using Python methods like append and extend to add elements.

We've now used two convenience methods from the Python class. It has many more, the most useful of which are listed in table 15.1.

Both CreateRuntime and CreateEngine optionally take a dictionary of options to configure the engine/runtime. GetSysModule, GetBuiltinModule, GetClrModule, and

Table 15.1 Methods on the Python class

Method name	Purpose
CreateRuntime(): ScriptRuntime	Creates a new ScriptRuntime with the IronPython scripting engine preconfigured.
CreateEngine(): ScriptEngine	Creates a new ScriptRuntime and returns the ScriptEngine for IronPython. If the ScriptRuntime is required, it can be acquired from the Runtime property on the engine.
GetEngine(ScriptRuntime runtime): ScriptEngine	Given a ScriptRuntime, this gets the ScriptEngine for IronPython.
GetSysModule(this ScriptEngine engine): ScriptScope	Gets a ScriptScope, which is the Python sys module for the provided ScriptRuntime.
GetBuiltinModule(this ScriptEngine engine): ScriptScope	Gets a ScriptScope, which is the Python __builtin__ module for the provided ScriptRuntime.
GetClrModule(this ScriptEngine engine): ScriptScope	Gets a ScriptScope, which is the Python clr module for the provided ScriptRuntime.
ImportModule(this ScriptRuntime runtime, string moduleName): ScriptScope	Imports the Python module by the given name and returns its ScriptScope. If the module does not exist, an exception is raised.

ImportModule can all be called with either a ScriptRuntime or a ScriptEngine. These four methods are also extension methods, so if you're using .NET 3.5 you'll be able to do someEngine.GetSysModule() after using or importing IronPython.Hosting.

Back to our example: as well as passing in the command-line arguments, we also need to make sure that the import path is set up correctly for the Python program.

SETTING THE SEARCH PATH

The Python engine has a method called SetSearchPaths, which allows us to pass in an array of paths. These set up sys.path, which is the module search path for import statements and adding references to assemblies.

In order to properly replicate the environment a program runs under when executed with ipy.exe, we need to honor the IRONPYTHONPATH environment variable. This means fetching the environment variable and breaking up a string like C:\Python25\lib;C:\Python\25\lib\site-packages into its component parts by splitting on the semicolons.

Perhaps *more* important, the directory the Python program is in needs to be in the import path. We can't assume that this is the current directory, because the application could have been launched from a shortcut or simply from another directory.

Listing 15.2 shows code that creates a List (a generic .NET List, not a Python one) and populates it with the directory containing the Python program plus any paths from IRONPYTHONPATH. It then calls engine.SetSearchPaths with an array from the List.

Listing 15.2 Populating the engine import paths from C#

```
string filename = "program.py";
string path = Assembly.GetExecutingAssembly().Location;     <-- Location of executing application
string rootDir = Directory.GetParent(path).FullName;

List<string> paths = new List<string>();
paths.Add(rootDir);

string path = Environment.GetEnvironmentVariable("IRONPYTHONPATH");
if (path != null && path.Length > 0)     <-- Check IRONPYTHONPATH for contents
{
   string[] items = path.Split(';');
   foreach (string p in items)
   {if (p.Length > 0) { paths.Add(p); }}
}
engine.SetSearchPaths(paths.ToArray());     <-- Set the paths
```

Listing 15.3 does the same thing, but in VB.NET.

Listing 15.3 Populating the engine import paths from VB.NET

```
Dim filename As String = "program.py"
Dim path As String = Assembly.GetExecutingAssembly().Location     <-- Location of executing application
Dim rootDir As String = Directory.GetParent(path).FullName

Dim paths As List(Of String) = New List(Of String)()
paths.Add(rootDir)

Dim path As String = Environment.GetEnvironmentVariable("IRONPYTHONPATH")
```

```
If path <> Nothing AndAlso path.Length > 0 Then      <-- Check IRONPYTHONPATH
   Dim items As String() = path.Split(";"c)              for contents
   For Each p As String In items
      If p.Length > 0 Then
         paths.Add(p)
      End If
   Next
End If
engine.SetSearchPaths(paths.ToArray())     <-- Set the paths
```

One thing to be careful of in VB.NET is that the `And` logical operator does not short circuit (it evaluates both sides of the expression even if the first is `False`). Before splitting the `IRONPYTHONPATH` environment variable, we need to check that it is not null *and* not empty. We need to use the `AndAlso` operator because we can't check the length of the string if it is null.

Now we're ready to execute the code, but with one caveat: because we are executing code in a dynamic language, we need to be able to handle runtime exceptions gracefully.

HANDLING PYTHON EXCEPTIONS

Our .NET code is just a thin wrapper around the Python program, so unhandled exceptions will be fatal anyway, and we could simply let them bubble up. The problem with this approach is that the traceback written to the console will be an incomprehensible CLR traceback, including all the stack frames inside the DLR. What we want is a straightforward Python traceback, and we can get this by catching the exception and using `ExceptionOperations` to format the traceback.

We have to confess to slightly misleading you earlier; although `Execute` is a perfectly valid way to run the code in a `ScriptSource`, it isn't the best way to run a script that is a full program. Python programs can exit with a return code[6] (an integer), and if this happens we want to propagate the exit code. Instead of `Execute`, we can use `ExecuteProgram`, which returns us the integer exit code (and will create its own scope rather than requiring us to pass one in). Listing 15.4 executes the `ScriptSource` we created from program.py inside some exception-handling code. If the program terminates normally, it exits with the return code from `ExecuteProgram`. If an exception is raised, it writes out the formatted exception message and exits with a return code of 1.

Listing 15.4 Executing the `ScriptSource`, handling any exceptions (in C#)

```
try
{
   ScriptSource source;
   source = engine.CreateScriptSourceFromFile(programPath);
   int result = source.ExecuteProgram();
   return result;
}
catch (Exception e)
{
   ExceptionOperations eo = engine.GetService<ExceptionOperations>();
```

[6] A Python program exits with an explicit return code by calling `sys.exit(integer)`.

```
    Console.Write(eo.FormatException(e));
    return 1;
}
```

We have pulled all this together into the `EmbeddedExecutable` example in the downloadable source code.

Something the example does that isn't shown in the code snippets above is to set debug mode on the engine. This is done by passing in a dictionary of options to the `CreateEngine` call, with `Debug` set to `true`. It causes the IronPython engine to compile Python code in debug mode (without optimizations—so it isn't recommended for production code). This allows you to use the Visual Studio debugger, including setting breakpoints, on the Python code.

When you run EmbeddedExecutable.exe, it runs program.py, which prints a message along with the command-line arguments it is passed. It then raises an exception to illustrate the exception formatting, as you can see in figure 15.2.

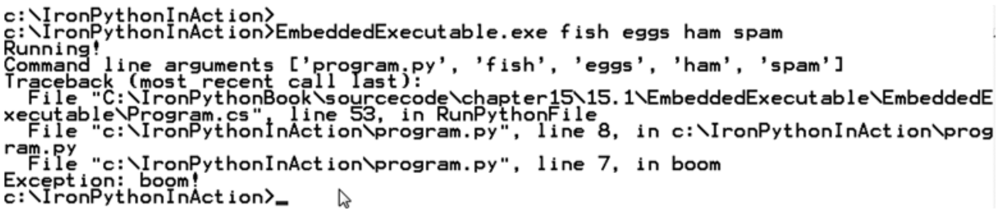


Figure 15.2 An executable application (.exe) that launches a Python program

Embedding IronPython in a custom executable uses only a fraction of the hosting API, but we have used some of the most important classes. The `ScriptEngine`, `ScriptScope`, and `ScriptSource` will accompany us through the rest of the book.

Next we look at some different hosting scenarios and explore more of the API. These build on what we have already learned, but instead of just executing Python code, we actually interact between the host application and the Python environment it is hosting.

15.2 IronPython as a scripting engine

By embedding IronPython in an application we can do much more than create executable wrappers over Python applications. Potential uses include providing scripting capabilities and plugins for .NET applications and even writing (or prototyping) parts of an application in IronPython.

In this section we work through another example,[7] which covers the core classes and basic techniques for embedding IronPython. In the next section we build on this with a more specific example that uses IronPython to add a plugin mechanism to a program.

Topics we cover in this section are creating compiled code objects from Python code, setting and fetching variables from execution scopes, adding references to

[7] We use the `BasicEmbedding` example in the downloadable sources.

assemblies, and publishing Python modules to runtimes. This will give us several different ways to interact between IronPython code and the .NET application. We can directly place objects into an execution context (a scope) for Python code to use—possibly calling back into our application from these objects. We can then fetch objects back out of the execution context after Python code has run. Alternatively, we can add references to assemblies or build and publish Python modules, so that IronPython code can access them by importing them.

15.2.1 Setting and fetching variables from a scope

The methods that initialize the main execution scope and then execute the Python code are very similar to the code we have already written. There are two important differences, though.

When we were running a Python program, we created a `ScriptSource` from the source code file and called `ExecuteProgram` to run it. `ExecuteProgram` executes code in its own execution scope. However, if we execute code in an explicitly created `ScriptScope`, we can then access variables, classes, and the like that have been created by the executed code. Conversely, if we want to provide objects for the code to use, we can place them in the scope prior to execution.

The second difference in the embedding code we are about to write is the way that code is executed. In the snippets we have just seen, a `ScriptSource` is created from the source code and then executed directly with the scope. If we are going to execute the code several times, which is entirely likely in a hosting situation, then we can optimize by compiling the code from the `ScriptSource`.

Calling `script.Compile()` returns us a `CompiledCode` object that we can use in place of the `ScriptSource`.

`ScriptScope` and `CompiledCode` are two highly reusable components. You can create multiple scopes and execute the same code in the different scopes or create a single scope and execute different code with access to the same objects.

The `ScriptScope` is a DLR class and is *not* tied to IronPython. You can execute Python code to create an object graph in a scope and then execute Ruby code in the same scope and with access to the same objects.[8] This permits some very interesting interoperability stories. Where IronRuby (or Managed JScript or IronScheme or …) uses IronPython objects, they retain their behavior as Python objects but are still usable from these other languages. There are some restrictions in the ways they can interoperate; it is unlikely that Python classes will ever be able to inherit from Ruby classes or vice versa, for example.[9] It should still be possible for dynamic languages running on .NET to share libraries, though. Python on Rails or Ruby on Django, anyone?

From a hosting point of view, the interesting thing we can do with scopes is to set variables in and fetch variables out of them. This means that you can publish an object

---

8 The Silverlight DLRConsole application does exactly this.

9 If they could, which metaclass should they use—the Python one or the Ruby one?

model from the hosting application into the scope for user code to work with (call methods, add handlers to events, and so on). After executing Python code you can then fetch objects it has created out of the scope.

> **Python scopes and variables**
>
> We use the term *variable* to refer to the contents of a Python scope with some unease. Although it is probably the right term when used from the .NET side, Python doesn't really have a concept of *variables*. Instead it has objects and names referencing those objects. Nonetheless, *variable* is the term used in the API for named objects in a `ScriptScope`.

The methods for working with names contained in a scope are shown in table 15.2.

Table 15.2 `ScriptScope` methods for working with names and variables

Method name	Purpose
`ClearVariables() : Void`	Clears all the variables in the scope.
`ContainsVariable(String) : Boolean`	Returns `true` if the scope contains the specified name.
`GetVariable(String) : Object`	Fetches the named variable from the scope as Object. Will raise an UnboundNameException if the variable does not exist.
`GetVariable<T>(String) : T`	Fetches the named variable as type T. Will raise an ArgumentTypeException if the variable is of the wrong type.
`RemoveVariable(String) : Boolean`	Removes the named variable, returning `true` for success.
`SetVariable(String, Object) : Void`	Sets the specified variable in the scope.
`TryGetVariable(String, out Object) : Boolean`	Takes an out parameter, which will not be set if the variable does not exist in the scope.

You also have read-only access to all the variable names as `IEnumerable` of string through the public `VariableNames` property, and the variables themselves as `IEnumerable` of `KeyValuePairs` (containing string, object pairs) through the `Items` property.

The most important of these methods are the three that allow you to set and fetch variables in the scope:

- `GetVariable`
- `TryGetVariable`
- `SetVariable`

Setting variables is straightforward. You call `SetVariable` with any string and any object. If you want Python code to actually have access to the variables you set, then

you should restrict the strings (variable names) to be valid Python identifiers, but the scope doesn't enforce this.[10]

Fetching variables has more complications associated with it, mainly because of the impedance mismatch of interacting with a dynamic language from a statically typed language. So long as the variable exists—and after executing arbitrary Python code there's no guarantee of that—C# and VB.NET *insist* on knowing the type before they will allow you to do anything useful with it.

For known types you have a choice of checking that the variable exists with `ContainsName` and use the generic version of `GetVariable` to fetch it from the scope. To fetch a string you use `GetVariable<string>(name)` from C# or `GetVariable(Of String)(name)` from VB.NET. Alternatively you can use `TryGetVariable`, which takes an out parameter that will be null (nothing) after the call if the variable doesn't exist.[11] From C# you will then need to cast the value to the known type after fetching it. `TryGetVariable` returns a Boolean indicating success or failure of attempting to fetch the variable.

If you are executing arbitrary code you will, therefore, need code that can handle the variable not existing *or* being the wrong type. If you are executing known code rather than arbitrary code, then it is fine to do any necessary error handling within the Python code and be able to guarantee that the variable exists and is of the expected type.

For .NET types, once you've pulled them out of the scope you have full access to them as if they were created from C#/VB.NET. If they are dynamic objects, such as functions or classes, you can still use them—but you have to use mechanisms provided by the Dynamic Language Runtime to perform operations on them.[12] This is something that we will look at later in the chapter.

In the meantime we now have all the pieces we need to set variables in a scope, execute code in that scope, and then fetch objects back out. Listing 15.5 is the start of an `Engine` class in C#. You instantiate it with Python source code as a string.

Listing 15.5 Creating an execution scope with access to contained variables (in C#)

```
public class Engine
{
   ScriptEngine _engine;
   ScriptRuntime _runtime;
   CompiledCode _code;
   ScriptScope _scope;

   public Engine(string source)
   {
      _engine = Python.CreateEngine();
      _runtime = _engine.Runtime;
```

[10] This mirrors the behavior of Python, where you can also programmatically set invalid identifiers in a namespace.

[11] Or if the variable is set to `None` inside the scope.

[12] This is not hard, but will be even easier once support for dynamic operations is built into the CLR or C#/VB.NET languages—as is happening in C# 4.0 and VB.NET 10.

```
        _scope = _engine.CreateScope();
        _scope.SetVariable("__name__", "__main__");          <-- Set name in scope

        ScriptSource _script = _engine.CreateScriptSourceFromString(source,
       SourceCodeKind.Statements);                            <-- Create ScriptSource from Python code
        _code = _script.Compile();       <-- Create CompiledCode from ScriptSource
    }

    public bool Execute()
    {
        try
        {
            _code.Execute(_scope);       <-- Execute Python code in the scope
            return true;
        }
        catch (Exception e)
        {
            ExceptionOperations eo = _engine.GetService<ExceptionOperations)();
            Console.Write(eo.FormatException(e));
            return false;
        }
    }

    public void SetVariable(string name, object value)
    {
        _scope.SetVariable(name, value);
    }

    public bool TryGetVariable(string name, out result)
    {
        return _scope.TryGetVariable(name, out result);
    }
}
```

The `Execute` method catches any exceptions raised in the Python code (writing the Python exception to standard out) and returns a Boolean indicating success or failure. The `SetVariable` and `TryGetVariable` methods on the engine merely delegate to methods on the `ScriptScope`. Handling error conditions, such as the variable not existing or being of the wrong type, is up to the caller.

This code does something that our last example didn't. Now that we know how to set variables in our execution scope, we can set the `__name__`. Python code would expect `__name__` to exist, and for the top-level script it would expect it to be set to `__main__`. The reason why we *didn't* do this when creating the executable is that `engine.CreateScriptSourceFromFile` implicitly sets the name in the scope to the name of the file (minus the .py).[13] The advantage of using `CreateScriptSourceFromFile` is that it doesn't read the whole file at once. If your top-level program *depends* on the name being set to `__main__`, then you can have the best of both worlds by using the three-argument form: `engine.CreateScriptSourceFromFile(path, Encoding.Default, SourceCodeKind.Statements)`.

This allows you to execute the `ScriptSource` in a scope with an explicit `__name__` set, without it being implicitly overridden because it is from a file.

[13] This is the same behavior when importing a module in Python; the name in the module corresponding to the file os.py is `os`, for example.

With what you know already you could probably achieve almost everything you need when embedding IronPython. Magically injecting variables is not (always) the most elegant way of exposing objects to your hosted code. A much nicer solution is to make your objects available either in Python modules or in assemblies that the Python code can import in the usual way.

15.2.2 Providing modules and assemblies for the engine

Adding references to assemblies is done with the `LoadAssembly` method on the `ScriptRuntime`. This takes an actual assembly object, so we need to use the `System.Reflection` API to obtain it.

We can use this to overcome another minor limitation when embedding IronPython. ipy.exe adds references to mscorlib.dll and System.dll (the core assemblies containing the `System` namespace) for us. This means that `import System`, or variants, can be executed without explicitly having to add references to these assemblies. Code running inside the default embedded IronPython engine will need to manually add the references before being able to import from `System`. We can solve this by getting Assembly objects from types in each of the two assemblies. We can do this in C# with this:[14]

```
_runtime.LoadAssembly(typeof(String).Assembly);
_runtime.LoadAssembly(typeof(Uri).Assembly);
```

Which translates to this in VB.NET:

```
_runtime.LoadAssembly(GetType(String).Assembly)
_runtime.LoadAssembly(GetType(Uri).Assembly)
```

The `BasicEmbedding` example also includes a ClassLibrary.dll assembly containing a class with a couple of static methods (shared functions in VB.NET–speak) that write to stdout. We can add a reference to this assembly by first loading it with `Assembly.LoadFile` from the same directory as the executable. This time we have the example code in VB.NET, shown in listing 15.6.

Listing 15.6 Adding a reference to an assembly on the `ScriptRuntime` (in VB.NET)

```
Dim _assembly As Assembly
Dim libraryPath As String

Dim fullPath As String = Assembly.GetExecutingAssembly().Location
Dim rootDir As String = Directory.GetParent(fullPath).FullName

libraryPath = Path.Combine(rootDir, "ClassLibrary.dll")     <- Path to assembly on disk

_assembly = Assembly.LoadFile(libraryPath)
_runtime.LoadAssembly(_assembly)
```

Once we've added a reference to the assembly, Python code running in the hosted engine is free to import from its namespace(s).

If the object model you want to expose isn't contained conveniently in a single assembly, or you want to construct it from live objects at runtime, then an alternative is to construct a Python module and add that to the runtime instead.

[14] The choice of Uri is entirely arbitrary. We just need some class that lives in System.dll.

In Python, import statements first check to see if the requested module is already in sys.modules. The embedded equivalent is runtime.Globals, and just like sys.modules it doesn't have to be a "real" Python module you put in there but any object you want Python code to be able to import.

ScriptRuntime.Globals is actually our old friend the ScriptScope. This means that you set objects in it using the same SetVariable method we have already used. You can put any object into there, and import statements executed in the embedded engine will fetch them. Using this technique is another way to make a host object model available to user code.

The challenge is that we *do* want to create a real Python module to put in there. We could just create and populate a new ScriptScope and publish that. Under the hood IronPython modules use a ScriptScope to store the namespace, and it would behave like a module when imported. However, ScriptScope doesn't have the right repr, and it has a few other minor differences, so it will seem a bit odd if the user does introspection on a ScriptScope published as a module. The answer is to use the Scope object, which as far as IronPython is concerned *is* a real Python module. We construct a Scope object from a ScriptScope using HostingHelpers.GetScope.[15]

The following snippet of VB.NET creates a ScriptScope called inner in which we set a string with the name HelloWorld. This is wrapped in a Scope object that is then published into the runtime globals.

```
Dim _module As Scope
Dim inner As ScriptScope

inner = _engine.CreateScope()
inner.SetVariable("HelloWorld", "Some string...")

_module = HostingHelpers.GetScope(inner)
_runtime.Globals.SetVariable("Example", _module)
```

Code running in the embedded engine can either execute import Example and access HelloWorld as a module attribute or execute from Example import HelloWorld to get direct access to the string we set in inner.

We've now covered all the major classes necessary for a wide range of different embedding scenarios. Figure 15.3 summarizes what we learned so far. It shows the core classes that we have worked with and the relationship between them and their useful members. Our core Engine class is now basically complete, with only one minor modification needed. Since we know how to make modules available for importing, we turn the scope in which we execute the main script into a proper module.

This C# snippet does this and puts the module into the runtime globals with the name __main__:

```
_scope = _engine.CreateScope();
_scope.SetVariable("__name__", "__main__");

Scope _main = HostingHelpers.GetScope(_scope);
_runtime.Globals.SetVariable("__main__", _main);
```

[15] HostingHelpers lives in the Microsoft.Scripting.Hosting.Providers namespace. ScriptScopes are "remotable wrappers" for Scopes. HostingHelpers.GetScope gets the local version, so it will work only for local ScriptScopes. There are other ways of creating Scopes when using the DLR remoting support.

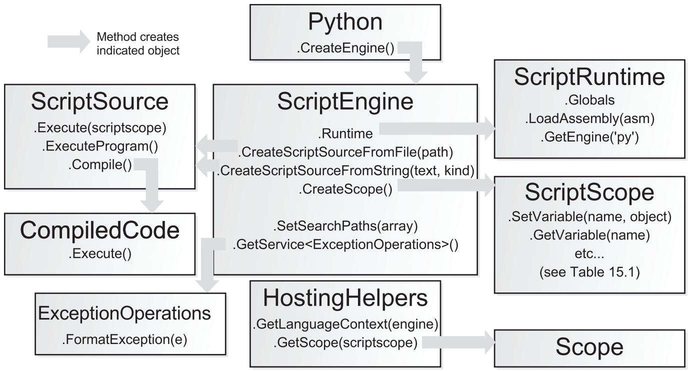


Figure 15.3 Core hosting classes[16]

The advantage of doing this is that the Python code `import __main__` now does the right thing. This is not a common thing to do but is used, for example, by `unittest.main()` to introspect the main execution scope and find all `TestCase` classes.

The example project uses the `Engine` class by embedding a Python script as a resource. Let's see how it uses the engine.

15.2.3 Python code as an embedded resource

This example executes a Python file that is stored as an embedded resource compiled into the main assembly. This can be a useful way of preventing Python code your application depends on from being modified in your distributed application. Even though it isn't distributed as plain text, it *isn't* an effective way of keeping the source code secret, as it will be easily discoverable within the assembly.[17]

Adding a Python source file as an embedded resource to a Visual Studio project is as simple as adding the file (either from an existing file or adding a new text file and renaming) and setting the Build Action to Embedded Resource, as shown in figure 15.4.

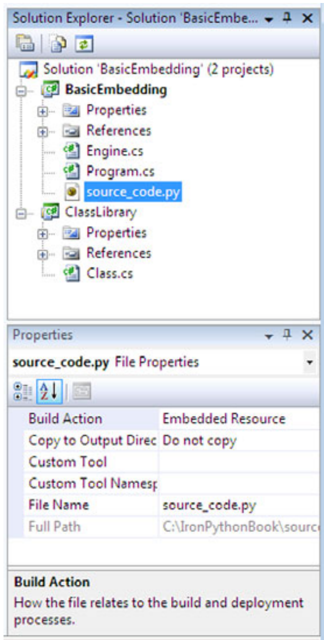


Figure 15.4 Creating embedded resources in Visual Studio

Listing 15.7 shows the C# code to retrieve the source code from the embedded resource as a string.

---

[16] Some of these classes have *other* useful members. This diagram is a reference to the ones we have used so far.

[17] Although you could encrypt your Python source code. Because that will require the means to decrypt the code in the assembly as well, it is still discoverable—but probably more work than assemblies compiled from C#, which are usually trivially disassembled with Reflector.

Listing 15.7 Fetching an embedded resource from an assembly (in C#)

```
static string GetSourceCode()
{
   Assembly assembly = Assembly.GetExecutingAssembly();
   string name = "BasicEmbedding.source_code.py";
   Stream stream = assembly.GetManifestResourceStream(name);
   StreamReader textStreamReader = new StreamReader(stream);
   return textStreamReader.ReadToEnd();
}
```

The important call here is `assembly.GetManifestResourceStream(name)`, which gives us access to our embedded source file as a stream.

Our example retrieves the source code and uses the `Engine` we have created to execute it. Before executing it sets a variable (imaginatively called *variable*) into the execution scope and fetches it out again after execution. Listing 15.8 shows the full code in VB.NET.

Listing 15.8 Using the `Engine` to execute Python code, with error handling (in VB.NET)

```
Sub Main()
   Dim source As String
   Dim engine As Engine

   source = GetSourceCode()                        <-- Fetch the Python code
   engine = New Engine(source)

   engine.SetVariable("variable", "Hello World!")

   Dim result As Boolean = engine.Execute()        <-- Execute the code

   If (Not result) Then
      Console.WriteLine("Executing Python code failed!")
   Else
      Dim variable As Object = Nothing
      Dim success as Boolean
      success = engine.TryGetVariable("variable", variable)   <-- Fetch "variable" from execution scope
      If (success) Then
         Console.WriteLine("""variable"" = {0}", variable)
      Else
         Console.WriteLine("Fetching ""variable"" failed")
      End If
   End If
End Sub
```

`engine.Execute` does the error handling for us, returning a Boolean indicating whether or not the execution succeeded. Assuming we *can't* trust our code not to fail, we need to check this result and handle the failure. Because we are actually executing code from an embedded resource and *not* arbitrary user code, we could skip the error checks if they aren't necessary. If the code executes correctly, it fetches the `variable` variable back out of the execution scope (handling the case when it doesn't exist in the scope) and writes it out to the console. Because we are writing it out to the

console, we fetch it as an `object`, neatly bypassing the problem of having to know what type it is.

Unfortunately, this type problem we skipped over is quite a big issue for using embedded IronPython for anything practical. Although we already have most of the pieces of the jigsaw, the next section puts them together (and shows one way of resolving the type problem) by implementing a plugin system for a .NET application.

15.3 Python plugins for .NET applications

The example we've just worked through has taken us through a good proportion of the IronPython hosting API. There are still some useful tricks we can learn, though. In particular we can solve the problem of how to communicate between dynamic languages and C#/VB.NET, which have to know the types of objects in order to be able to do anything useful with them.

The goal of this example is to create a .NET application that allows the user to write plugins that extend the functionality in Python. The application interface consists of a toolbar and a multiline textbox.

The user can add plugins by creating Python files in the application's plugins directory, which are all loaded when the application starts. Any plugin that is successfully loaded has a button added to the toolbar, and the plugin is called—with access to the application textbox—when its button is clicked. The final product is shown in action in figure 15.5.

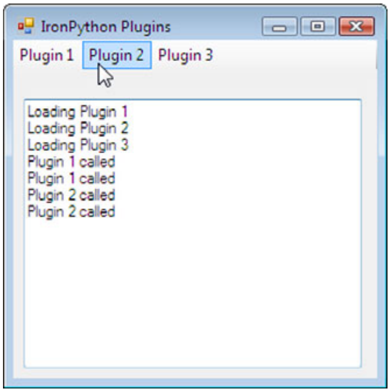


Figure 15.5 The IronPython Plugins example application

Although this is a trivially simple example, it solves the real-world problems involved in creating a user-extensible application. It loads an arbitrary number of Python plugin scripts and provides them with an API in a clean manner. As well as bridging between code written in a dynamically typed language and an application written in statically typed languages, we will also be covering some useful techniques for plugin situations. These include

- The autodiscovery of plugins, dynamically executing them at runtime rather than requiring configuration or other mechanisms
- Diverting the standard output and error streams for hosted DLR runtimes and their engines
- Handling specific exceptions from hosted Python code

We start by looking at how to create the host environment so that Python code can simply and cleanly add plugins.

15.3.1 A plugin class and registry

In order to solve the type problem, we can provide a base class that plugin classes must inherit from. On the .NET side we can handle user plugins as this base class, and the compiler is happy—knowing what methods and properties are available.[18]

We also have to provide a mechanism for the plugins to be added to the application. We can do this with a `PluginStore` class, also accessible to user code, which acts as a registry for plugins.

Our `PluginBase` class is instantiated with a name, which will be used for the toolbar button. It also provides an `Execute` method, which does nothing on the base class but will be called with the textbox in real plugins (when the corresponding toolbar button is clicked).

On the .NET side, where we need to interact with the user plugins, we can use the `PluginBase` type. Listing 15.9 shows an implementation in C#.

Listing 15.9 A base class for user plugins to inherit from (in C#)

```
public class PluginBase
   {
      private string _name;
      public string Name
      {
         get { return _name; }
      }

      public PluginBase(string name)          <- The constructor takes a name
      { _name = name; }

      virtual public void Execute(TextBox textbox)     <- Plugins override this method
      { }
   }
```

The `PluginStore` class is equally simple. It has a public static (shared) method, `AddPlugin`, which takes a `PluginBase` instance. This adds the plugin to a list accessed as the `Plugins` property. This list is marked as internal (friend), which means that only classes in the same assembly can access the list and it isn't exposed to user code. Listing 15.10 shows the `PluginStore` class in VB.NET.

Listing 15.10 The `PluginStore` registry class (in VB.NET)

```
Public Class PluginStore

   Private Shared _plugins As List(Of PluginBase) _
      = New List(Of PluginBase)()

   Friend Shared ReadOnly Property Plugins() _
      As List(Of PluginBase)
      Get
         Return _plugins
      End Get
```

[18] Additional user-defined methods and members won't be directly visible, of course.

```
    End Property

    Public Shared Sub AddPlugin(ByVal plugin As PluginBase)
        _plugins.Add(plugin)
    End Sub
End Class
```

In fact, the `PluginStore` is the *only* public class in the main assembly. We can add the main assembly to the runtime (along with the assembly containing the `PluginBase`), and user code can call `PluginStore.AddPlugin` without being able to access any of the application internals that we haven't explicitly exposed.

Listing 15.11 shows the Python code that creates a new plugin and adds it to the `PluginStore`.

Listing 15.11 IronPython plugin using `PluginBase` and `PluginStore`

```
from Plugins import PluginBase
from EmbeddingPlugin import PluginStore

class Plugin(PluginBase):
  def Execute(self, textbox):
    textbox.Text += "Plugin 1 called\r\n"

plugin = Plugin("Plugin 1")
PluginStore.AddPlugin(plugin)
```

Our application can execute all the user scripts and then access any successfully added plugins via `PluginStore.Plugins`. Let's look at how our application loads the user scripts.

15.3.2 Autodiscovery of user plugins

Our application uses a very simple mechanism to load user scripts. Alongside the executable is a directory called plugins. When the application starts, it executes all the Python files in this directory.

As we are executing user code we obviously need to be tolerant of errors, and to make things more interesting, let's see how we can handle specific Python errors differently.

The `SyntaxErrorException` is raised when we attempt to execute invalid code. It lives in the `Microsoft.Scripting` namespace. It is an exception shared by all DLR languages. The Python-specific exceptions live in the `IronPython.Runtime.Exceptions` namespace, so with the right `using` or `Imports` directive we can catch specific Python errors. Listing 15.12 shows the code that loads and executes all the plugins. Any errors in executing plugin code are caught, and a message box is displayed to the user, but syntax errors and `SystemExit` exceptions[19] are treated differently with a custom message. Figure 15.6 shows the error message shown to users for syntax errors.

Listing 15.12 is the C# code that loads and executes the scripts from the plugins directory.

[19] This is raised if the user calls `sys.exit(n)`.

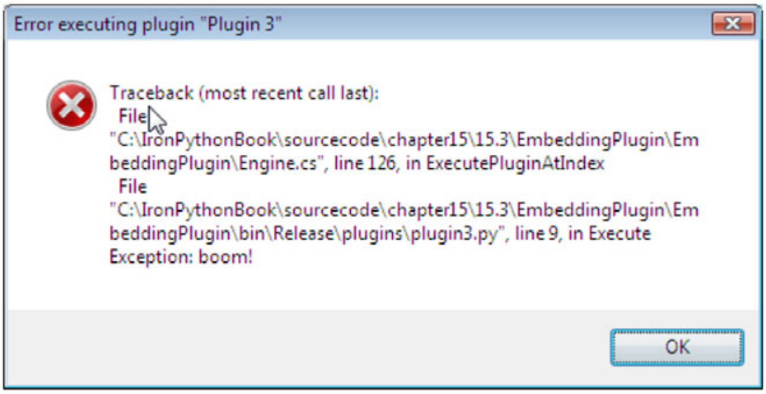


Figure 15.6 Custom error message for syntax errors in plugin code

Listing 15.12 Finding and executing user plugins (in C#)

```
using IronPython.Runtime.Exceptions;

public void LoadPlugins()
{
   string exePath = Assembly.GetExecutingAssembly().Location;
   string rootDir = Directory.GetParent(exePath).FullName;
   string pluginsDir = Path.Combine(rootDir, "plugins");      <-- Find the plugins directory

   foreach (string path in Directory.GetFiles(pluginsDir))
   {
      if (path.ToLower().EndsWith(".py"))       <-- Filter the Python files
      {
         CreatePlugin(path);
      }
   }
}

public void CreatePlugin(string path)
{
   try
   {
      ScriptSource script;
      script = _engine.CreateScriptSourceFromFile(path);
      CompiledCode code = script.Compile();
      script.Execute();
   }
   catch (SyntaxErrorException e)              <-- Catch Python syntax
   {
      string msg = "Syntax error in \"{0}\"";
      ShowError(msg, Path.GetFileName(path), e);
   }
   catch (SystemExitException e)               <-- Catch SystemExit
   {
      string msg = "SystemExit in \"{0}\"";
      ShowError(msg, Path.GetFileName(path), e);
   }
   catch (Exception e)
   {
```

```
            string msg = "Error loading plugin \"{0}\"";
            ShowError(msg, Path.GetFileName(path), e);
        }
    }

public void ShowError(string title, string name, Exception e)
{
    string caption = String.Format(title, name);
    ExceptionOperations eo = _engine.GetService<ExceptionOperations>();
    string error = eo.FormatException(e);
    MessageBox.Show(error, caption, MessageBoxButtons.OK,
        MessageBoxIcon.Error);

}
```

The `ShowError` method may seem slightly odd: the caller has to extract the filename from the full path instead of it being done inside the method. It is written this way because we reuse `ShowError` to show a different kind of message when we call the plugins.

Before we finish off the application by hooking up the plugins to the toolbar buttons, we look at how we can improve the development process for the user writing the plugins.

15.3.3 Diverting standard output

This example is a Windows Forms application, which makes for much better screenshots. It does have one disadvantage, though; it means that print statements inside the user code are lost because there is no standard output for them to go to. This could make debugging plugins much harder for our users. Luckily, DLR runtimes support diverting the standard output and error streams, and in this example we send them to the application textbox.

The API for diverting the output and error streams[20] is through `runtime.IO.SetOutput` and `runtime.IO.SetError` methods that take a .NET `Stream` and an encoding. To use these methods, we need a `Stream` that diverts everything written to it back to the textbox.

`Stream` is an abstract class—inheriting from it is easy but requires implementing a tedious number of methods and properties. For this example we've chosen to inherit from `MemoryStream` and override the `Write` method. This is an abuse of `MemoryStream`, but it works fine.

Listing 15.13 shows the `PythonStream` class and setting an instance onto the runtime to divert both standard output and standard error.

Listing 15.13 Diverting output streams to a textbox (in C#)

```
internal class PythonStream: MemoryStream
{
    TextBox _output;
    public PythonStream(TextBox textbox)
    {
```

[20] We could also do this from within Python code by replacing `sys.stdout` and `sys.stderr` with custom objects.

```
        _output = textbox;
    }

    public override void Write(byte[] buffer, int offset,
                               int count)
    {
        string text = Encoding.UTF8.GetString(buffer, offset,
            count);
        _output.AppendText(text);
    }
}

public void SetStreams()
{
    PythonStream stream = new PythonStream(_box);
    _runtime.IO.SetOutput(stream, Encoding.UTF8);
    _runtime.IO.SetErrorOutput(stream, Encoding.UTF8);
}
```

Minor points are worth noticing from this listing. When we set the streams on the runtime we need to specify an encoding; so in our custom stream we need to use the *same* encoding to decode the bytes back into a string. The stream is constructed with a textbox, and the `Write` method simply calls `AppendText` to add new text.

The "Loading Plugin 1" and other messages shown in figure 15.5 actually come from print statements in the plugin code. When the plugin is loaded (executed), anything written to standard out appears in the main textbox.

The final thing we need to do is to create a toolbar button per plugin and hook up its `Click` event to call the appropriate `Execute` method.

15.3.4 Calling the user plugins

Once the plugins have all been loaded, we can iterate over `PluginStore.Plugins` and add a button and handler for each one. The code to do this from C# is shown in listing 15.14.

Listing 15.14 Adding toolbar buttons and click handlers per plugin (in C#)

```
int index = 0;
foreach (PluginBase plugin in PluginStore.Plugins)
{
    ToolStripButton button = new ToolStripButton();
    button.ToolTipText = plugin.Name;
    button.Text = plugin.Name;

    int pluginIndex = index;
    button.Click += delegate {
        ExecutePluginAtIndex(pluginIndex);
    };

    pluginToolStrip.Items.Add(button);
    index++;
}

public void ExecutePluginAtIndex(int index)
{
```

```
    PluginBase plugin = PluginStore.Plugins[index];

    try
    {
        plugin.Execute(_textbox);
    }
    catch (Exception e)
    {
        string msg = "Error executing plugin \"{0}\"";
        ShowError(msg, plugin.Name, e);
    }
}
```

In C# we can use a closure to call the right plugin when its handler is invoked. By turning the index into a variable local to the scope of the `foreach` loop (the `pluginIndex` variable), the anonymous delegate will invoke `ExecutePluginAtIndex` with the correct index when its button is clicked.

The VB.NET code, shown in listing 15.15, has to be slightly different.

Listing 15.15 Adding toolbar buttons and click handlers per plugin (in VB.NET)

```
Dim index As Integer = 0
For Each plugin As PluginBase In PluginStore.Plugins

   Dim button As ToolStripButton = New ToolStripButton()
   button.ToolTipText = plugin.Name
   button.Text = plugin.Name

   Dim handler As ButtonHandler
   handler = New ButtonHandler(engine, index, button)
   pluginToolStrip.Items.Add(button)
   index += 1
Next

Friend Class ButtonHandler
   Dim _engine As Engine
   Dim _index As Integer

   Public Sub New(ByVal engine As Engine, _
              ByVal index As Integer, _
              ByVal button As ToolStripButton)
      _engine = engine
      _index = index

      AddHandler button.Click, AddressOf ClickHandler
   End Sub

   Public Sub ClickHandler(ByVal sender As Object, _
                    ByVal e As EventArgs)
      _engine.ExecutePluginAtIndex(_index)
   End Sub
End Class

Public Sub ExecutePluginAtIndex(ByVal index As Integer)
   Dim plugin As PluginBase = PluginStore.Plugins(index)

   Try
```

```
        plugin.Execute(_box)
    Catch e As Exception
        Dim msg As String = "Error executing plugin ""{0}"""
        ShowError(msg, plugin.Name, e)
    End Try
End Sub
```

Because we don't have the luxury of closures in VB.NET, we have to use a class to capture the state (which is what the compiler does under the hood for us in C# anyway). For every plugin a new `ButtonHandler` instance is created, which has a `ClickHandler` to be called when the button is used.

In the previous two sections of this chapter we have gone through two different examples of embedding IronPython. The first was a gentle amble through the hosting API that introduced us to all the major classes. The example that we have just completed showed us how to use embedded IronPython for a practical purpose, in the form of application plugins. This example solved the mismatch between dynamic and statically typed languages by using a specific type for plugins. We can also use more general techniques to interact with dynamic objects and, in the process, extend IronPython to some new use cases.

15.4 Using DLR objects from other .NET languages

One of the (many) great advantages of Python is that code can be modified without having to be recompiled. It is stored as plain text and can even be generated at runtime. There has been a lot of focus in the technical world recently on using dynamic languages to provide Domain Specific Languages (DSLs), small languages that model a particular problem domain.

We can use IronPython for this purpose by evaluating expressions or executing code stored as text or received as user input. Business rules can be stored in text files and edited without the application needing to be recompiled.

This use case requires new ways of interacting with dynamic objects. We need to be able to create and use normal Python objects (the built-in types and Python classes and instances) from .NET, while retaining their behavior as Python objects.

15.4.1 Expressions, functions, and Python types

The simplest example of this is to use IronPython to evaluate individual expressions. IronPython makes a great calculator![21]

So far, whenever we have executed Python code from a string we have passed in the enumeration member `SourceCodeKind.Statements`. This member has a sister, `SourceCodeKind.Expression`, that allows us to evaluate an expression and return an object. It is used in this snippet of C# to evaluate a simple mathematical expression:

```
string code = "2 + 3 + 5";
ScriptSource source;
```

---

[21] You can see an example of embedded IronPython as a calculator at http://www.voidspace.org.uk/ironpython/dlr_hosting.shtml.

```
source = runtime.CreateScriptSourceFromString(code,
   SourceCodeKind.Expression);
int result = source.Execute<int>(scope);
```

This code uses the generic form of `Execute` that allows us to specify the type of the returned object. Using the generic version requires us to pass in an explicit scope to `Execute`. If we wanted to use the default scope, we could instead cast the returned object to retrieve the integer.

> **SourceCodeKind and interactive sessions**
>
> The `SourceCodeKind` enumeration has another member[22] as well: `InteractiveCode`. This can detect incomplete statements and is useful for executing code from interactive sessions, from a `TextBox` acting as a console in a UI, for example. You can call `ScriptSource.GetCodeProperties()`, which returns a `SourceCodeProperties` value that will tell you if the source code is invalid or incomplete.

Since we are retrieving results with specific types, we can use the `System.Func` delegate to define functions in Python and call them from other .NET languages. This delegate is a standard part of .NET 3.5 (C# 3.0), but it is also provided by IronPython 2, so you can happily use it in .NET 2.0 projects.

The simplest example of this is to evaluate an expression that returns us a lambda function, as shown in listing 15.16.

Listing 15.16 Creating and using Python functions from C#

```
string code = "lambda x, y: x * y";

ScriptSource source;
source = engine.CreateScriptSourceFromString(code,
   SourceCodeKind.Expression);

Func<int, int, int> lambda;
lambda = source.Execute<Func<int, int, int>>(scope);

int result = lambda(9, 8);
Console.WriteLine("9 * 8 = {0}", result);
```

Listing 15.17 shows the equivalent VB.NET code.

Listing 15.17 Creating and using Python functions from VB.NET

```
Dim code As String = "lambda x, y: x * y"

Dim source As ScriptSource
Dim lambda As Func(Of Integer, Integer, Integer)
source = engine.CreateScriptSourceFromString(code, SourceCodeKind.Expression)

lambda = source.Execute( _
```

[22] Actually it has two more. The purpose of `SourceCodeKind.SingleStatement` speaks for itself.

```
Of Func(Of Integer, Integer, Integer))(scope)
Dim result As Integer = lambda(9, 8)
Console.WriteLine("9 * 8 = {0}", result)
```

`Func` is generic, with type specifiers for the arguments and the return value. This example specifies a function that takes two integers as arguments and returns an integer: `Func<int, int, int>`. The last type specified is always the return type. There are convenient definitions for functions that take up to eight arguments (and if that isn't enough, you need to rethink your function API!).

Instead of evaluating an expression to create a lambda, we could execute a function definition in a scope and pull the function out by name. This would work with any callable object, and we could pull them out as any form of delegate—allowing the `ScriptScope` to do the language conversion when we pull out the value.

As well as allowing you to use dynamically created functions, you could use Python functions to provide a .NET API to Python libraries.

We can use functions because they have a corresponding type defined in C#, and the same is true of the other Python types. The built-in Python types live in the `IronPython.Runtime` namespace. The very basic types, like strings and integers, *aren't* there because IronPython uses the standard .NET types for these. Types that are available include the following:

- `List`
- `FrozenSetCollection`
- `PythonDictionary`
- `PythonFile`
- `PythonGenerator`
- `PythonTuple`
- `SetCollection`

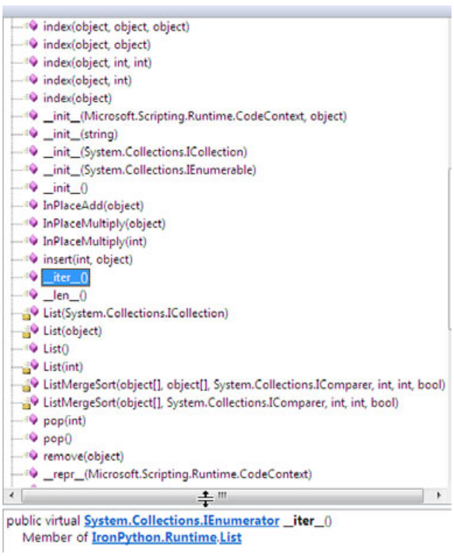


Figure 15.7 The Python List type viewed from the Visual Studio 2008 Object Browser

Most of the normal Python methods are available on these types, including the magic double-underscore methods. You can explore the types and their members using Visual Studio 2008 Object Browser or Reflector.[23] Figure 15.7 shows some of the members on the Python List type.

We can retrieve these objects by type from a Python scope or create and use them directly. When we created a custom executable for a Python program, we created a Python List and set it on the language

[23] This extremely useful tool for introspecting .NET assemblies is available at http://www.red-gate.com/products/reflector/.

context as `sys.argv`. Listing 15.18 shows creating a tuple in Python and then using it from C#.

Listing 15.18 Creating a tuple in Python and using it from C#

```
string code = "('hello', 'world')";
ScriptSource source;
source = engine.CreateScriptSourceFromString(code,
   SourceCodeKind.Expression);

PythonTuple tuple = source.Execute<PythonTuple>(scope);
Console.WriteLine("tuple = ({0}, {1})", tuple[0], tuple[1]);
```

These techniques are great for the built-in Python types, and for objects with known types, but they don't allow us to use classes defined in Python. Fortunately, there are other techniques we can use with Python classes and instances.

15.4.2 Dynamic operations with ObjectOperations

IronPython (and IronRuby) classes are *not* .NET classes. IronPython classes are .NET objects—instances of `IronPython.Runtime.Types.PythonType`, the Python `type` metaclass.[24] This is because Python classes can have members added and removed at runtime. You can even change their base classes dynamically, things that we couldn't do with .NET classes.

Python classes are still usable from C# and VB.NET; we can actually solve the type problem by ignoring it! We can keep objects as objects on the .NET side and use the Python engine to perform operations like creating instances and calling methods.

The mechanism for doing this is to use `ObjectOperations`, a class that provides dynamic operations for DLR objects. The `ScriptEngine` exposes this as the `Operations` property, returning an `ObjectOperations` instance bound to the semantics of the engine's language.

`ObjectOperations` is a class with many useful methods, and again the best way to find out what it can do is to explore it with Reflector or the Visual Studio Object Browser. It knows how to perform comparisons and mathematical operations on dynamic objects, following the semantics of the language for those operations. In the case of Python that means autopromoting integers to longs if necessary and using the `__add__` method for addition where appropriate, and so on.

More important, `ObjectOperations` knows how to call objects and set and fetch members. Using these capabilities alone we can achieve most of what we might want to do with dynamic objects. Listing 15.19 shows how to fetch a Python class out of an execution scope, create an instance of that class, and call a method on it.

Listing 15.19 Creating class instances and calling methods using `ObjectOperations`

```
string code = @"
class Something(object):
   def method(self, value):
```

[24] All Python classes are instances of their metaclass, and for new style classes the base metaclass is `type`. `type` can also be used as a function to tell you the type of objects.

```
    return value + 1
";

ObjectOperations ops = engine.Operations;
ScriptSource source;
source = engine.CreateScriptSourceFromString(code,
   SourceCodeKind.Statements);
source.Execute(scope);              <-- Create the class

object klass = scope.GetVariable("Something");     <-- Fetch the class object

object instance = ops.Call(klass);                 <-- Create an instance
object method = ops.GetMember(instance, "method");

int result = (int)ops.Call(method, 99);     <-- Call a method
Console.WriteLine("99 + 1 = {0}", result);
```

Calling the method is a little cumbersome, but if we were working with known Python classes, it would be easy to abstract away with a thin wrapper layer.

We create an instance of a class in the same way we do in Python, by calling the class. This is done with `ops.Call`, and afterward we fetch a method from the instance with `ops.GetMember`. `GetMember` takes the instance and the name of the member we want to fetch as a string. This mirrors the way we use the `getattr` function from Python.

Having fetched the method we call it, again with `ops.Call`. Any arguments passed to `ops.Call` are passed in as arguments to the object we are calling; the overload we are using is `Call(object, params object[])`. We cast the objects to known types only at the point at which we actually need them, in this case the return value from calling the method. `GetMember` also has a generic version, which we can call if we care about the type of object it will return.

The corresponding partner to `GetMember` is `SetMember`. Its signature matches `setattr`; `ops.SetMember(object, name, value)` where `name` is a string specifying the member to set.

Table 15.3 lists methods of `ObjectOperations`, including the three we have already discussed.

Table 15.3 Methods on `ObjectOperations` for working with dynamic objects

Method name	Purpose
`Call`	Calls the object with any arguments specified
`GetMember`	The equivalent of the two-argument form of `getattr`
`SetMember`	The equivalent of `setattr`
`RemoveMember`	The equivalent of `delattr`
`GetMemberNames`	Lists all members; the equivalent of `dir`
`ContainsMember`	The equivalent of `hasattr`

Table 15.3 Methods on `ObjectOperations` for working with dynamic objects *(continued)*

Method name	Purpose
`TryGetMember`	Takes an out parameter that will be left unmodified if the member does not exist
`ConvertTo/TryConvertTo`	Converts a dynamic object to the specified type, either in generic form or taking a `System.Type`

It may not be immediately obvious why we might need `ConvertTo` and `TryConvertTo` when we can always cast objects. We can use them to take advantage of some of the magic that IronPython does on our behalf, particularly when using Python functions as event handlers. Under the hood IronPython wraps the function as a delegate for us, and we can use `ObjectOperations.ConvertTo` to do the same from the .NET side. The following snippet shows how to do this from C#:

```
ObjectOperations ops = engine.Operations;
object function = scope.GetVariable("function");
SomeObj.SomeEvent += ops.ConvertTo<DelegateType>(function);
```

Remember that if you want to be able to unhook the handler, you'll need to keep a reference to the delegate it returns!

As well as being able to fetch and call methods on dynamic objects, we can use the Python built-in functions on them.

15.4.3 The built-in Python functions and modules

If we have a collection of objects created from Python code and we want to add them up, we have several choices of how to do it. We could set them in an execution scope with names and evaluate an expression that adds them up, or we could use `ObjectOperations.Add` in a loop. We have a third choice as well; Python has a perfectly good built-in function for adding things up: `sum`.

Python has a wide range of useful built-in functions, and in IronPython they are implemented as static methods on `IronPython.Runtime.Builtin`. Some of the built-in functions require a `CodeContext`; we'll see how to provide this when we look at working with built-in modules. `sum` is perhaps not the most useful example to pick, but some of the built-in functions are very useful for working with dynamic objects.

For example, if we have obtained (by whatever means) a dynamic object that we believe to be an instance of a particular class, we can confirm this by using the `isinstance` function. Listing 15.20 shows an example of using both the `isinstance` and the `issubclass` functions from C#.

Listing 15.20 Using the built-in functions `isinstance` and `issubclass` from C#

```
string code = @"
class Something(object):
   def method(self, value):
```

```
        return value + 1

class SomethingElse(object):
    pass
";

ObjectOperations ops = engine.Operations;

ScriptSource source;
source = engine.CreateScriptSourceFromString(code,
    SourceCodeKind.Statements);
source.Execute(scope);

PythonType klass;
PythonType klass2;
klass = scope.GetVariable<PythonType>("Something");
klass2 = scope.GetVariable<PythonType>("SomethingElse");

object instance = ops.Call(klass);

bool isinstance = Builtin.isinstance(instance, klass);
bool isinstance2 = Builtin.isinstance(instance, klass2);
bool issubclass = Builtin.issubclass(klass, typeof(object));
bool issubclass2 = Builtin.issubclass(klass, klass2);

Console.WriteLine("isinstance(instance, Something) = {0}",
              isinstance);                                  <-- True
Console.WriteLine("isinstance(instance, SomethingElse) = {0}",
              isinstance2);                                 <-- False
Console.WriteLine("issubclass(Something, object) = {0}",
              issubclass);                                  <-- True
Console.WriteLine("issubclass(Something, SomethingElse) = {0}",
              issubclass2);                                 <-- False
```

This example creates two Python classes and fetches them both out of the scope. It creates an instance of the first one (`Something`) and then performs various checks on the instance and the classes using `Builtin.isinstance` and `Builtin.issubclass`.

One difference about the way we use Python classes in this example is that we are specifically pulling them out as `PythonType` rather than `object`. `isinstance` takes two objects, but `issubclass` requires the first argument to be a `PythonType` (new style class) or `OldClass` (guess).

We've now used the built-in types and the built-in functions, but there is one more class of Python built-in that we haven't used directly with embedded IronPython. From the title of this section it won't come as a shock to you to hear that these are the built-in Python modules.

The built-in modules are implemented in the `IronPython.Modules` assembly. In there you will find a single entry for each module (`ClrModule`, `PythonDateTime`, `PythonSocket`, and so on) plus one for each type that the module exports. Functions in the module are defined as static methods on the module object.

Listing 15.21 uses the Python `pickle` module[25] to serialize and deserialize a Python dictionary. It then checks that the serialization and deserialization have worked, using

[25] See http://docs.python.org/lib/module-pickle.html.

ObjectOperations.Equal to compare the original dictionary and the deserialized object. In order to use the pickle loads and dumps functions, we must first create a CodeContext.[26]

Listing 15.21 Serializing and deserializing Python objects from C# with pickle

```
string code = "{'name': 'Michael Foord', 'Age': 21, 'Profession': 'Software
     Development'}";

ObjectOperations ops = engine.Operations;

ScriptSource source;
source = engine.CreateScriptSourceFromString(code,
   SourceCodeKind.Expression);
object dict = source.Execute(scope);

LanguageContext language = HostingHelpers.GetLanguageContext(engine);
CodeContext co = new CodeContext(new Scope(), language);        <- Create a CodeContext
string cereal = (string)PythonPickle.dumps(co, dict, 0, null);  <- Serialize
object dict2 = PythonPickle.loads(co, cereal);                  <- Deserialize
bool result = ops.Equal(dict, dict2);

Console.WriteLine("original and unmarshalled dictionaries equal = {0}",
     result);
```

The call to PythonPickle.dumps takes a CodeContext, a Python object, a protocol, and a bin argument. First we construct a CodeContext with a new Scope and the appropriate LanguageContext. Creating CodeContexts like this is useful for working with several of the built-in functions and modules.

This example uses protocol 0, which ensures that the resulting pickle is serialized as ASCII rather than using a binary format. Storing binary data in Unicode .NET strings is a recipe for pain, and it means we can pass in null for the bin argument.

dumps returns an object, which we can cast to a string. If we feed this string back into PythonPickle.loads (along with the context), it deserializes the string back into a Python dictionary for us. This could be useful for persisting state from the dynamic part of an application.

The DLR hosting API is larger than we could hope to cover in a single chapter. Topics we haven't had a chance to look at include the Dynamic Language Runtime support for creating engines and executing code inside AppDomains. This is a good way of limiting what user code is able to do when run inside your application. It also has a Platform Adaptation Layer (PAL), used by Silverlight, that allows you to customize the way files are opened and imports are resolved and even to execute code in custom namespaces that can provide or change names on demand.

Not only that, but there is an enormous number of ways you could use embedded IronPython. What we have managed to do, though, is look at some of the most common ways of using IronPython, and most of the numerous alternatives will bear *some* similarity to the scenarios we have explored. What I really hope you take away from

[26] CodeContext lives in the Microsoft.Scripting.Runtime namespace.

this chapter is a kick-start in being able to imagine what is possible and knowing how to start exploring. Before we finish, there is some big news about how you will interact with dynamic languages in future versions of the .NET framework.

15.4.4 The future of interacting with dynamic objects

At the 2008 Microsoft PDC conference, Jim Hugunin and Anders Hejlsberg announced[27] that the DLR would be integrated into version 4.0 of the .NET framework. Alongside this there will be changes to C# and Visual Basic to introduce dynamic features that use the DLR.

The major change in C# 4.0, at least the one that is relevant to us, is the addition of the `dynamic` keyword. This is a static declaration to the compiler that operations on the object are to be handled dynamically at runtime! Operations on objects declared as dynamic will be delegated to the DLR. For ordinary .NET objects the DLR uses reflection (just as it does inside IronPython), but it also enables some things not normally possible from C#. These include duck typing, the use of late-bound COM, and the creation of fluent APIs,[28] like XML or DOM traversal using element names as object attributes.

More important, it allows you to receive objects from a DLR language engine and use them as dynamic objects. Objects created by IronPython or IronRuby can be used from C# while retaining their behavior as Python and Ruby objects.

The C# Future documentation[29] gives this example of using the new `dynamic` keyword. All of the uses of d shown here will be done by the DLR:

```
dynamic d = GetDynamicObject(...);
d.M(7); // calling methods
d.f = d.P; // getting and settings fields and properties
d["one"] = d["two"]; // getting and setting through indexers
int i = d + 3; // calling operators
string s = d(5,7); // invoking as a delegate
int a = d; // assignment conversion
```

The final operation creates a typed object (`a`) from the dynamic object `d`. As with the examples you've been working on in this chapter, these operations could raise runtime errors, and so the sort of error handling that we've been discussing will still be needed.

The bottom line is that the hosting APIs make it easy to work with dynamic languages, but interacting with dynamic objects will get a whole lot easier.

---

27 Jim's blog has links to videos of their talks: http://blogs.msdn.com/hugunin/archive/2008/10/29/dynamic-language-runtime-talk-at-pdc.aspx.

28 .NET objects that implement the `IDynamicObject` interface can provide custom behavior when used dynamically.

29 This documentation is available from http://code.msdn.microsoft.com/csharpfuture/.

15.5 Summary

The integration of IronPython with the underlying .NET runtime is what makes it such a joy to work with. This extends to embedding IronPython in C# and VB.NET. Creating and using language engines is straightforward (once you know the magic incantations), so experimenting is easy and enables things that previously might have required writing your own scripting language!

There is a natural complication when using objects created by a dynamic language from statically typed languages; this is something that can and *will*[30] get easier, but there are many ways you can minimize the intricacy. One useful principle is to do as much as possible inside the engine. If you do your type checking and error handling in Python, then you can guarantee to return a known type into your statically typed code. When you really *want* to work with a dynamic object, then `ObjectOperations` is your friend.

We've now used IronPython from both the inside and the outside, and we've also reached the end of the last chapter. This willingness to experiment, and an excitement about the possibilities, is the most important message of the book. Have fun programming!

[30] For example, see the discussion about C# 4.0 at http://channel9.msdn.com/posts/Charles/C-40-Meet-the-Design-Team/.

appendix A: A whirlwind tour of C#

Obviously this book has primarily been about IronPython, but in several places we've explored some aspects of using IronPython with other .NET languages, the foremost being C#. If you don't know C#, this appendix will give you a brief overview of the core parts of the language. It should be enough to let you read the code examples in the book, although you'll probably want more detail if you're trying to write C# code.

A.1 Namespaces

Namespaces are similar to Python packages and modules. The classes in an assembly live in namespaces, which can be nested to provide a hierarchical structure. This structure is used to group together related classes to make them easier to find. Namespaces are specified in C# like so:

```
namespace EvilGenius.Minions {
    class GiantRobot {
    ...
    }
}
```

Here we declare a namespace called `EvilGenius.Minions`, which contains the `GiantRobot` class. Another major feature of namespaces is that they enable us to avoid name collisions: the fully qualified name of the class is `EvilGenius.Minions.GiantRobot`, and we could happily use other `GiantRobot` classes in our project as long as they are in different namespaces.

This is similar to how Python modules work, but namespaces differ from modules in several ways. First, namespaces are entirely independent of the name of the file in which they are defined, and you can have a file that defines multiple namespaces. Conversely, you can define classes in a particular namespace in multiple different files. For example, if the `GiantRobot` class is defined in `GiantRobot.cs`, you could

have another file defining the fully qualified class `EvilGenius.Minions.CorruptBureaucrat`. Both classes are in the `EvilGenius.Minions` namespace.

Namespaces and assemblies are orthogonal; assemblies are the physical organization of code, whereas namespaces provide logical organization.

A.2 Using directive

In C# code, you have access to any classes in the current assembly or referenced assemblies. Classes in the current namespace can be referred to directly. To refer to classes in other namespaces, you can use fully qualified class names in your code, but that's very verbose. The alternative is to add a *`using` directive* to the top of the file:

```
using EvilGenius.Minions;
```

Then occurrences of the unadorned class name `GiantRobot` in the code refer to `EvilGenius.Minions.GiantRobot`.

In the case of a name collision (say you also wanted to use `Transformers.Autobots.GiantRobot` in the code), the `using` directive can create an alias:

```
using autobots=Transformers.Autobots;
```

Then you can refer to the alternative `GiantRobot` class as `autobots.GiantRobot`.

A.3 Classes

Class declarations in C# are structured like this:

```
public class GiantRobot: Robot, IMinion, IDriveable {
    public string name;
    public GiantRobot(string name) {
        this.name = name;
    }
    // more fields and methods here
}
```

The *access modifier* `public` before the `class` keyword indicates that the class is accessible outside the current namespace. The names after the colon specify what this class inherits from. `GiantRobot` is a subclass of the `Robot` class, as well as the `IMinion` and `IDriveable` interfaces. C# doesn't allow multiple inheritance of classes, but implementing multiple interfaces is fine. If no parent class is specified in a class definition (or it inherits only from interfaces), the class inherits from `System.Object`.

The body of the class can contain definitions of various kinds of members:

- Constructors
- Fields (like the `name` attribute in the previous snippet)
- Methods
- Properties
- Events
- Operators

Each of these members has access modifiers as well:

- `private`–Can be used only inside this class
- `protected`–Accessible from this class and any subclasses
- `internal`–Accessible from any class in the current assembly
- `public`–Available from any code

Classes can also contain other types (including classes, interfaces, structs, or delegates) as members.

As well as access modifiers, other modifiers can be applied to classes when they are defined:

- `abstract`–This declares that the class can be used only as a base class and can't be instantiated. It might contain declarations of abstract methods (without any implementation) that subclasses must define when they are declared. (Abstract classes can still contain concrete implementations of some methods.)
- `static`–This indicates that this class can't be instantiated or used as a type (so it can't be subclassed). Static classes are essentially containers for their members; in Python it would make more sense to use a module.
- `sealed`–This prevents this class from being subclassed, for security or performance reasons. When the compiler sees a method call on a sealed class, it can generate a direct call to the method, rather than a virtual call.
- `partial`–Partial classes are classes that are defined in more than one file. This can be useful if some code in a class is machine generated; the generated code can be kept in a separate file and regenerated without clobbering the custom code.

A.4 Attributes

Attributes are declarative information that can be attached to different elements of a program and examined at runtime using reflection. They can be applied to types, methods, and parameters, as well as other items. The .NET framework defines a number of attributes, such as `SerializableAttribute`, `STAThreadAttribute`, and `FlagsAttribute`, which control various aspects of how the code runs. You can also create new attributes (to be interpreted by your own code) by inheriting from `System.Attribute`.

The following code applies the `SecretOverrideAttribute` to the `OrbitalWeatherControlLaser.Zap` method:

```
class OrbitalWeatherControlLaser {
    //...
    [SecretOverride("zz9-plural-z-alpha")]
    public void Zap(Coords target) {
        //...
    }
}
```

As you can see, attribute constructors can take arguments, and the `-Attribute` suffix of the class name can be omitted when attaching it.

A.5 Interfaces

An *interface* is a type that defines methods and properties with no behavior. Its purpose is to make a protocol explicit without imposing any other restrictions on the implementation. For example, the previous `GiantRobot` class implements the `IDriveable` interface:

```
interface IDriveable {
    IDriver Driver {get; set;}
    void Drive();
}
```

This means that `GiantRobot` must provide a definition of the `Driver` property and the `Drive` method. Classes that implement multiple interfaces must provide definitions for all of the interfaces' members. Interfaces can subclass other interfaces (and even inherit from multiple interfaces), which means that they include all of the members of their bases as well as the members they declare.

Ordinarily, a class that inherits from an interface can implement its members simply by defining them with the correct name, parameters, and return type. In some cases, you might not want the interface members to be part of your class's interface (for example, if you have two different interfaces that clash). You can implement those members explicitly, so that they are available only when used through a reference to the interface:

```
public class ExplicitlyDriveable: IDriveable {
    // member prefixed with interface name
    public void IDriveable.Drive() {
        // implementation here...
    }
    // other members
}
```

Then code trying to use the `Drive` method on an `ExplicitlyDriveable` instance must cast it to `IDriveable` first.

A.6 Enums

Enumerations are collections of named constants, which look like this:

```
enum RobotColor {
    MetallicRed,
    MetallicGreen,
    MetallicBlue,
    Black
}
```

The enum values can be referred to in code using `RobotColor.MetallicBlue`. By default, enumerations inherit from `int`, but they can be declared (using the inheritance syntax) as any of the integral .NET types: `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, or `ulong`.

Enums can be combined as flags with the bitwise-or operator (|) if you annotate them with the Flags attribute, in which case you should also specify the value of each member (by default, they're assigned sequentially from 0). For example, if you wanted the RobotColor values to be combinable, you could define it as follows:

```
[Flags]
enum RobotColor {
    MetallicRed = 1,
    MetalicGreen = 2,
    MetallicBlue = 4,
    Black = 8
}
```

A.7 Structs

Structs are data structures that are defined similarly to classes. The difference is that a struct is a *value type* rather than a *reference type* like a class. When a variable is of a class type, it contains a reference to an instance of the class (or null). A struct variable directly contains the members of the struct rather than being a reference to an object elsewhere in memory. This is generally useful for performance or memory optimization: a large array of structs can be much quicker to create than the same size array of objects. In some situations using structs can be slower, though, particularly when assigning or passing them as parameters (because all their member data needs to be copied).

Struct definitions look like this:

```
struct Projectile {
    public float speed;
    public Projectile(float speed) {
        //...
    }
}
```

Structs can't inherit from any type (other than object, which they inherit from implicitly), and no types can inherit from them.

A.8 Methods

Methods in C# are members of classes and structs that are defined by giving the access level, the return type, the name of the method, and the parameters it accepts, as well as the block of code that is executed when it is called:

```
class GiantRobot {
    //...
    public void GoToTarget(string targetName) {
        Point location = this.targetDatabase.Find(targetName);
        this.FlyToLocation(location);
    }
}
```

The void return type indicates that this method returns nothing.

A.8.1 Virtual and override methods

The virtual modifier applied to a method indicates that it can be overridden in a subclass. The subclass's version of the method must have the same visibility, name, parameter types, and return type as the base class method and be defined using the keyword override.

```
class GiantRobot {
    //...
    public virtual void TravelTo(Point destination) {
        this.TurnTowards(destination);
        while (this.Location != destination) {
            this.Trudge();
        }
    }
}

class FlyingRobot: GiantRobot {
    //...
    public override void TravelTo(Point destination) {
        if (this.WingState == WingState.Operational) {
            this.FlyTo(destination); // whee!
        } else {
            base.TravelTo(destination); // oh well...
        }
    }
}
```

(Methods defined with override can also themselves be overridden by subclasses.)

In a method, this refers to the current instance in the same way as self in Python, although this doesn't need to be a parameter of each method. To call a superclass method or property (for example, from the body of an override method), you refer to base, which works similarly to calling super(Class, self) in Python, although the semantics are simpler because any C# class has only one superclass.

A.8.2 Other method modifiers

A number of other modifiers can be included after the access modifier when defining a method. The more commonly seen modifiers are these:

- *static*—This method must be called directly on the class rather than on an instance (and so can't refer to this). Often code that would be in a standalone function in Python is put into a static method in C#.
- *sealed*—This override method can't be overridden again in a subclass. This is often done for performance or security reasons.
- *abstract*—This method must be implemented in subclasses. Abstract methods can't contain any code.

A.8.3 Parameter passing

By default, method parameters are *passed by object value* in the same way as in Python (with the extra wrinkle that structs are copied). In Python, this is the only way to pass

parameters; if you want to see modifications by a function or method, you need to mutate an object that is passed in.

In C# you can change this behavior using the `ref` and `out` parameter modifiers. If a parameter has the `ref` modifier, it is *passed by reference* instead. Assignments to that parameter in the method will be reflected in the variable passed in by the caller. For example, an `int` variable passed as a `ref` parameter can be assigned in the method, and the variable's value will be updated in the calling scope. `Out` parameters are similar, with the difference that they don't need to be assigned a value before calling the method, and they *must* be assigned in the method. The difference between them is that `out` parameters are viewed as extra return values of the method (for which you might use a tuple in Python), while `ref` parameters are values that are both input *and* potentially output of the method.

A.8.4 Method overloading

There can be multiple different methods in a class with the same name, as long as the combination of name and parameter types is unique within the class. Methods that have the same name but different parameters are said to be *overloaded*. When you call an overloaded method, the compiler uses the compile-time types of the parameters passed in to determine which version of the method is called. This is an important distinction between overload dispatch, where the selection of which overload to call is purely based on the declared types of the parameters, and method dispatch, where the method called is determined by the type of the instance at runtime, regardless of the declared type of the variable.

A.9 Delegates

A *delegate* is a type that represents a reference to a method. Delegates are essentially the same as Python function references; a delegate that has been created from an instance carries a reference back to that instance in the same way as a bound method reference. Delegate definitions look like this:

```
delegate int IntegerFunction(int a, int b);
```

This creates an `IntegerFunction` type that accepts two integers and returns an integer.

In addition to creating delegates from method references, you can create them from anonymous functions, which are very similar to Python lambdas:

```
IntegerFunction f = (int a, int b) => (a * a + b + 1);
```

A.10 Events

An *event* is a member of a class that enables it to notify other code when something happens. Each event is declared with a delegate type, and interested parties add delegates of that type to the event; the delegates will be called when the event is triggered. For example, the `HideoutEntrance` might need to let other parts of the system know when there's an intruder:

```
delegate bool SecurityHandler(string details);
class HideoutEntrance {
    //...
    event SecurityHandler IntruderDetected;
}
```

Then interested parties can create `SecurityHandler` delegates and attach them to the `IntruderDetected` event so that they can respond to the problem:

```
hideoutEntrance.IntruderDetected += this.ActivateSentryTurret;
```

Event handlers can be removed with the `-=` operator. Events are triggered by calling them, in the same way as invoking a delegate (although if no handlers are attached, the event will be `null`).

A.11 Operator overloading

Operator overloading enables you to define how instances of your class should behave when they're used in expressions. For example, if you want to be able to add instances together, you can define `operator +`:

```
class Magnitude {
    //...
    public static operator +(Magnitude a, Magnitude b) {
        return Magnitude(a.size + b.size);
    }
}
```

These operator overrides work in much the same way as the magic methods in Python (such as `__add__`, `__mul__`, or `__eq__`). Operator members must be declared as `public static`, and they can be for unary operators (like `++`) as well as binary operators (like addition).

A.12 Properties and indexers

Properties are function members that are run when a particular attribute is retrieved or assigned. They can be used for validating the value that's being set or lazily creating an expensive attribute only when it's needed. A property definition looks like this:

```
class Heist {
    //...
    private float executionTime;
    public float Hours {
        get {
            return this.executionTime / 3600;
        }
        set {
            this.executionTime = value * 3600;
        }
    }
}
```

A property can be made read-only by leaving out the setter or made write-only by omitting the getter. The setter is called with an implicit parameter named `value`, which, surprisingly enough, is the value that was set.

Indexers are members that allow a class to act like a collection, similarly to the Python `__getitem__` and `__setitem__` magic methods. They can have getters and setters in the same way as properties, but they are defined slightly differently:

```
class LootBag {
    // container for items
    private ArrayList _underlyingItems;
    //...
    public object this[string name] {
        get {
            return this._underlyingItems[name];
        }
        set {
            this._underlyingItems[name] = value;
        }
    }
}
```

Indexers can be overloaded if you want to handle different types or numbers of parameters. In the same way as for properties, the value to be set is an extra implied parameter called `value`, and either the getter or setter can be omitted to make the indexers read- or write-only.

A.13 Loops

C# has four *looping* constructs: `while`, `do`, `for`, and `foreach`. `while` and `foreach` are like the `while` and `for` loops in Python; the `do` and `for` loops don't have any Python analogue.

A.13.1 while loop

The `while` loop works in almost exactly the same way as its Python counterpart. The only major difference is that the condition of the loop in C# must evaluate to a `bool`, while Python will allow any type.

```
int a = 10;
while (a > 0) {
    a--;
}
```

The loop control keywords `break` and `continue` work (for all C# loops) exactly the same way as they do in Python. The condition of the loop must be included in parentheses.

A.13.2 do loop

The `do` loop is like a `while` loop, where the condition is checked at the end of the loop body; that is, the loop body always executes at least once.

```
int a = 0;
do {
    a--;
} while (a > 0);
```

After the execution of this loop, `a` will have the value -1.

A.13.3 for loop

The C# `for` loop has three parts in addition to the loop body:

- An initialization clause, which sets up variables before the loop begins
- A condition that is tested before each execution of the loop body
- An iteration clause, which is executed after each execution of the loop body

Here's an example:

```
int total = 0;
for (int i = 0; i < 10; i++) {
    total += i;
}
```

After execution, `total` will have the value 45 (the sum of 0 to 9), because it stops when `i` equals 1. Each of the three control clauses is optional, and each clause can consist of multiple statements separated by commas.

A.13.4 foreach loop

`foreach` loops are the closest equivalent to Python's `for` loops. The `foreach` loop can iterate over any collection that implements the `IEnumerable` interface, including arrays.

```
string[] names = new string[] {"Alice", "Bob", "Mallory"};
foreach (string person in names} {
    Console.WriteLine(person);
}
```

A.14 Casts

Casts are used to convert an expression to a different type. There are two kinds of casts in C#:

```
GiantRobot robot = new GiantRobot();
IDriveable vehicle = (IDriveable)robot;
IMinion minion = (robot as IMinion);
```

The difference between these casts is that the first form, `(IDriveable)robot`, will raise an exception at runtime if the conversion is invalid, while the `robot as IMinion` expression will return `null` if there is no way to convert the type.

Casting from a class to one of its base classes or interfaces (*upcasting*) always succeeds. *Downcasting* (going in the opposite direction) can fail, because not every `IMinion` object is a `GiantRobot`.

A.15 if

The C# *`if`* statement looks very similar to the Python one. The condition must be in parentheses and yield a Boolean.

```
if (!robot.AtDestination()) {
    robot.Walk();
} else if (robot.CanSee(target)) {
    robot.Destroy(target);
```

```
} else {
    robot.ChillOut();
}
```

If statements can be chained as you see here, so there's no need for the elif from the Python if statement.

A.16 switch

Some situations that would require an if-elif-else construct or a dictionary lookup in Python can be done more cleanly in C# using a *switch* statement.

```
WeaponSelection weapon;
switch (target.Type) {
    case TargetType.Building:
        weapon = WeaponSelection.FistsOfDoom;
        break;
    case TargetType.ParkedCar:
        weapon = WeaponSelection.Stomp;
        break;
    case TargetType.FluffyKitten:
    case TargetType.FluffyBunny:
        weapon = WeaponSelection.MarshmallowCannon;
        break;
    case default:
        // death ray not operational yet
        weapon = WeaponSelection.InconvenienceRay;
        break;
}
```

The case labels must be constants, and each case has to end with a flow-control statement like break, throw, or return; execution isn't allowed to fall through from one case to another. If you want fall-through behavior, you can use the goto case statement to explicitly jump to the case that should be executed. Two case labels together (like the FluffyKitten and FluffyBunny case shown previously) are treated as if they share the same body. If the switch expression doesn't match any of the case labels and there's a default case, it is executed.

A.17 try/catch/finally and throw

Exceptions in C# are handled in the same way as they are in Python, with only minor syntactic differences:

```
try {
    robot.ReturnToBase(TransportMode.FootThrusters);
} catch (FootThrusterFailure) {
    robot.ReturnToBase(TransportMode.BoringWalking);
} catch (EmergencyOverrideException e) {
    robot.SelectTarget(e.NewTarget);
} catch {
    robot.AssumeDefensivePosition();
    throw; // re-throw the original exception
} finally {
    robot.NotifyBase();
}
```

In this example, `catch` is equivalent to Python's `except`, and `throw` corresponds to `raise`.

A.18 lock

All reference types in C# can be used as locks to prevent multiple threads from occupying critical sections at the same time. The *lock* statement provides a convenient way to specify a critical section:

```
lock(this.RightEye) {
    this.RightEye.Close();
    this.RightEye.Open();
}
```

This will prevent any other thread from executing this section of code with the same `RightEye` object. As long as other operations on the right eye are locked, it will also prevent this wink from being interrupted or interrupting some other operation.

The `lock` statement guarantees that the lock will be released when execution leaves the block, even if an exception is thrown.

A.19 new

The *new* operator is used to create new instances of types. It can be used to create class or struct instances, arrays, and delegates.

```
GiantRobot robot = new GiantRobot("destructo", true);
```

For classes and structs, the constructor that matches the parameters is called.

The `new` operator can also initialize the object that is created, by assigning to members (if the object is a class or struct instance) or by specifying the elements for a collection or array.

```
string[] demands = new string[] {"helicopter", "money", "island"};
```

When creating an array, if initial items have been provided, you don't need to also specify the size.

A.20 null

Where Python has `None`, a value that is the singleton instance of `NoneType`, the equivalent in C# is `null`. *null* is a reference that (paradoxically) doesn't refer to any object. This has two consequences: first, `null` can be assigned to a variable of any reference type, and second, value types like `int`, `DateTime`, or custom `structs` can't be `null`, because they don't contain references.

The second fact can be inconvenient in some situations, so each value type in C# has a *nullable* version, represented with the type name followed by a question mark. For example, a nullable integer variable would be declared with `int?`. Nullable types can then be used as normal (although there are performance differences under the hood) and be assigned `null` in the same way as reference types.

A.21 using statement

The *using* statement (not to be confused with the using directive) ensures that an object will be disposed of when execution leaves the block. It's essentially a specialized version of a try-finally statement.

```
using (DatabaseConnection conn = context.OpenConnection()) {
    // use the opened connection
}
// the connection will be Disposed at the end of the block
```

The target object must implement the IDisposable interface, which has only one member, the Dispose method. The using statement guarantees that Dispose will be called on the target, regardless of whether execution leaves the block by returning, throwing an exception, or falling off the bottom.

A.22 Operators

There's a large overlap between the *operators* in Python and C#. Table A.1 lists those that are different in C#:

Table A.1 Differences between C# and Python operators

Operator	Name	Description
`!`	logical negation	The equivalent of Python's `not` operator.
`++ --`	increment and decrement	Adds or subtracts 1 from a value. These are unusual from a Python perspective, because they both yield a value *and* mutate the operand, which can be confusing. Increment and decrement can be prefix or postfix; if prefix, the increment/decrement is done before yielding the value; if it is postfix, the increment or decrement is done after yielding the value.
`== !=`	equality and inequality	Generally used for identity (reference equality, `is` in Python) with reference types, although they can be overridden to be value equality. For value types, these are the same as the Python equivalents.
`is`	is	Corresponds to the Python `isinstance` function, although it can compare to only one type.
`&& ||`	logical and, logical or	Corresponds to Python `and` and `or`, and short-circuits in the same way. In the standard implementations the result is converted to `bool`, unlike Python's Boolean operators.
`??`	null coalescing	The expression `a ?? b` returns a, or b if a is null. A common Python idiom is to use `a or b` for this purpose, which works because `or` doesn't convert its result to a `bool`.
`?:`	conditional operator	The expression `a ? b : c` returns b if a is true, and c otherwise. It corresponds to the Python expression `b if a else c`.

A.23 Generics

Generic types are types that include one or more *type parameters*. They're often used to create containers or data structures that can operate on a range of types of objects without having to cast them to some common base type (in the most general case, this would be `object`). When you create an instance of a generic type, you specify the type parameters, and the instance acts as if those types had replaced all of the type parameters in the definition.

```
class LinkedList<T> {
    private T item;
    private LinkedList<T> rest;
    public LinkedList(T item) {
        this.item = item;
        this.rest = null;
    }
    public T Head {
        get { return this.item; }
    }
    public void InsertAfter(T newItem) {
        LinkedList<T> newNode = new LinkedList<T>(newItem);
        newNode.rest = this.rest;
        this.rest = newNode;
    }
    // more list methods...
}
```

This code (partially) defines a `LinkedList` generic class that could hold any object. The difference between this and an `ArrayList` or Python list (which could also hold any type of object) is that client code can create a `LinkedList` of `strings`, and then the compiler will ensure that only `strings` go in and come out.

```
LinkedList<string> list = new LinkedList<string>("abc");
list.InsertAfter("def");
string s = list.Head // no casting is required.
```

The `LinkedList` class doesn't need to do anything with the type parameter `T`, other than store instances of `T` or pass them around. Some generic classes need to have more interaction with their type parameters, for example, being able to create instances or call methods on the objects. To guarantee that the actual type used with the generic class has the necessary methods, generic classes can specify constraints on the type parameters, such as these:

- *`where T: class`*—Specifies that the type specified for `T` must be a reference type, while `where T: struct` constrains `T` to be a value type
- *`where T: <class or interface name>`*—Says that `T` must inherit from the class or interface
- *`where T: new()`*—Requires that `T` has a `no-argument` constructor
- *`where T: U`*—Specifies that `T` must be the same type or a subclass of the `U` type parameter

Any type constraints are checked at compile time when an instance of a generic type is created.

appendix B: Python magic methods

Creating your own objects in Python inevitably means implementing one or more of Python's protocol methods—the *magic methods* whose names start and end with double underscores. These protocols are roughly the equivalent of interfaces in Python.

The lookup rules for the magic methods are slightly different from those of other attributes. Normal attribute lookup order[1] is *instance -> class -> base classes*. Magic method lookup goes directly to the class, skipping the instance. This means that you can't override these methods by attaching a function directly to the instance (or through `__getattr__`); overriding has to happen at the class level. To provide these methods for classes themselves, they need to be implemented on the classes' class, that is, its metaclass.[2]

This appendix is a reference to all the common magic methods.[3]

---

1 This order assumes the usual caveat that the descriptor protocol makes the full lookup rules more complex. Section B.9 of this appendix describes the descriptor protocol.

2 The IronPython generics support using the `Array[int]` syntax, which is implemented by adding a `__getitem__` method to the `type` metaclass.

3 This is only a summary; for full details refer to the Python documentation at http://docs.python.org/index.html.

B.1 Object creation

The object creation methods are called when a class is instantiated, as shown in table B.1.

Table B.1 Object creation

Method	Description
`__new__(cls, ...)`	The object constructor. Responsible for creating new instances. It receives the class as the first argument, followed by all arguments passed in the instantiation call. To customize instance creation when inheriting from the built-in immutable types or .NET objects, you should override `__new__` instead of `__init__`.[a] This method returns the new instance, but there is no requirement for it to be an instance of the class provided. If the instance returned is not an instance of `cls`, then `__init__` will not be called.
`__init__(self, ...)`	The object initializer. Called after `__new__` when a new instance is created. Like other instance methods, it receives the instance as the first argument. It also receives all the arguments passed in the instantiation call. Returning anything other than `None` from this method will cause a `TypeError` to be raised.

a. Initializing immutable values is done in `__new__`, because otherwise it would be possible to change an immutable value by calling `__init__` with a different value.

B.2 Comparison

The rich comparison methods are called during comparison operations involving the `==`, `!=`, `<`, `<=`, `>`, and `>=` operators. They are also called by operations that implicitly involve comparisons, such as sorting a list. All these methods should return a Boolean. They can also return the `NotImplemented` singleton to indicate that the operation is not implemented for the two values being compared.

In Python there is no comparison fallback. To support all the comparison operators you need to implement all the comparison methods, shown in table B.2.

Classes of immutable objects that implement the comparison methods should also implement `__hash__`. Objects that compare equal should have the same hash.

Table B.2 Comparison methods

Method	Description
`__eq__(self, other)`	Called for equality operations (==). Python defaults to identity comparison for equality where `__eq__` is not defined.
`__ne__(self, other)`	Called for inequality operations (!=). Python defaults to identity comparison for inequality where `__ne__` is not defined.
`__lt__(self, other)`	Called for less-than operations (<).
`__le__(self, other)`	Called for less-than or equals operations (<=).

Table B.2 Comparison methods *(continued)*

Method	Description
`__gt__(self, other)`	Called for greater-than operations (>).
`__ge__(self, other)`	Called for greater-than or equals operations (>=).

B.3 Miscellaneous

Table B.3 provides a selection of important methods that don't fit into any other category.

Table B.3 Miscellaneous methods

Method	Description
`__nonzero__(self)`	Called to implement truth value testing and the built-in function `bool`. This method should return a Boolean. If a class doesn't implement `__nonzero__` but does implement `__len__`, then that will be called instead. A nonzero return value from `__len__` indicates that an instance is `True`. If a class defines neither `__len__` nor `__nonzero__`, all its instances are considered `True`.
`__subclasses__(self)`	Available on classes only. Returns a list of all subclasses of the class.
`__call__(self, ...)`	Called when an object is called as a function.
`__hash__(self)`	Called when an object is used as a dictionary key, for set membership, and by the built-in function `hash`. Should return a 32-bit integer or a `long` (new in Python 2.5). Objects that compare equal should have the same hash. Mutable objects (which should not be used as dictionary keys) should implement `__hash__` to raise a `TypeError`. It is easiest to implement `__hash__` by using the `hash` function on all the components of the object that play a part in comparison (either combining them with exclusive `or` or hashing a tuple of them).
`__del__(self)`	Called when a Python object is about to be destroyed. `__del__` can resurrect an object by creating a new reference to the object. A lot of the standard Python documentation for `__del__` does not apply, because IronPython uses .NET garbage collection rather than reference counting.

B.4 Containers and iteration

Python has two protocols for objects that support subscription (indexing): the sequence protocol and the mapping protocol. The sequence protocol[4] is used for sequences (like lists, tuples, and strings) that are indexed with an integer. The mapping protocol[5] is used for types (like dictionaries) that map keys to values. Both sequences and mapping types are typically iterable, with sequences iterating over their values and mapping types iterating over their keys.

4 For additional methods commonly defined by mutable sequence types see http://docs.python.org/lib/typesseq-mutable.html.

5 For additional methods commonly defined by mapping types see http://docs.python.org/lib/typesmapping.html.

B.4.1 Mapping and sequence protocol methods

Both the mapping and sequence protocols use __getitem__, __setitem__, and __delitem__ to fetch, set, and delete items. The mapping and sequence protocol methods are shown in table B.4.

Table B.4 Mapping and sequence protocol methods

Method	Description
__getitem__(self, index)	Called when an item or slice is fetched. If an item is indexed using slice syntax, then the index will be a slice[a] object. Classes may raise a TypeError if the index is an inappropriate type. A class implementing the sequence protocol should raise an IndexError if the index is outside the bounds of the sequence. A class implementing the mapping protocol should raise a KeyError instead. The same rules about slicing and exceptions also apply to __setitem__ and __delitem__. Note that these methods should also accept negative indices to mean indexing from the end (where 0>N>=-len(S)).
__setitem__(self, index, value)	Called when an item or slice is set.
__delitem__(self, index)	Called when an item or slice is deleted.
__len__(self)	Called by the built-in function len and returns the number of items in the container. May also be called when creating an iterator from an object.
__contains__(self, item)	Called by the in operator (x in y). Should return a Boolean indicating whether the container contains the item.
__iter__(self)	Called when iterating over an object or by the built-in function iter. It should return an iterator object. Iterator objects have an __iter__ method that returns self and a next method that returns the members of the iterator sequentially. Once the iterator is exhausted, subsequent calls to next should raise StopIteration.
__reversed__(self)	Containers may implement this method to provide an optimized reverse iterator for instances. If this method is available, it is called by the built-in function reversed.
__missing__(self, key)	New in Python 2.5. This method is called on subclasses of the built-in dictionary (dict) type when a key that doesn't exist is requested. This method is useful for providing default values.
__length_hint__(self)	Iterators can implement this as an optimization so that Python can preallocate space for them. IronPython doesn't use it (yet, anyway), but if you see this method, now you know what it is for.

a. The start, stop, and step attributes correspond to the [start:stop:step] components of the slice (or None if the component is omitted).

B.4.2 Generator expressions and conditional expressions

Generator expressions are a form of iteration that we haven't covered elsewhere. We've shown how list comprehensions allow for an extremely concise form of iteration and filter in a single expression:

```
result = [x for x in iterable if some_condition(x)]
```

This is comparable to LINQ over objects, which was introduced in C# 3.0 (.NET 3.5). There is an alternate form of list comprehensions called generator expressions, which arrived in Python 2.4. Instead of square brackets, they use parentheses to surround the expression. The major difference is that instead of being evaluated immediately, they are evaluated lazily. A generator expression returns a generator object (the same kind of object returned by a generator function). Like other iterators, this can be consumed by iterating over it, or you can consume individual items by calling the `next` method.

Because they are lazily evaluated, we can combine them:

```
from os import listdir
python_files = (f for f in listdir('.') if f.endswith('.py'))
first_lines = ((f, open(f).readline()) for f in python_files))
names_to_first_line = dict(first_lines)
```

None of the generator expressions are actually executed until the final `dict` call. A nice side effect is that they look nicer when used in places that take any iterable:

```
total = sum(val for val in some_list if val > 0)
```

Both list comprehensions and generator expressions can take advantage of another feature new to Python 2.5: conditional expressions (also known as *ternary* expressions).

Conditional expressions have the basic syntax

```
X if Y else Z
```

This evaluates `Y`, and if it is `True` it returns `X`—otherwise, it returns `Z`. We can use this in list comprehensions and generator expressions:

```
generator = (f(a) if test(a) else g(a) for a in some_list)
```

This generator expression tests each member of `some_list`, yielding `f(a)` if `test(a)` returns `True` and `g(a)` if `test(a)` returns `False`.

Advanced tools for working with iterators and generators can be found in the standard library module `itertools`; see http://docs.python.org/library/itertools.html.

B.5 Conversion to string

There are several ways that objects can be converted to strings, both implicitly and explicitly. Table B.5 lists the protocol methods that do this.

Table B.5 String conversion methods

Method	Description
`__repr__(self)`	Called by the built-in `repr` function and when an object is displayed on the console. This method should return a string. Many of the built-in types have string representation that can be used to reconstruct the object if passed in to `eval`.
`__str__(self)`	Called by the built-in `str` function (type) or when an object is printed. This method should return a string. If `__str__` is not defined on an object but `__repr__` is, then `str` will call this instead (so if you define only one, choose `__repr__`).
`__unicode__(self)`	Normally called by the built-in `unicode` function (type) and should return a Unicode string. In IronPython the `unicode` type is the `str` type, so `__str__` will be called instead.

B.6 Attribute access

In Python you can completely customize attribute access on objects for fetching, setting, and deleting. Table B.6 shows the attribute access methods.

Table B.6 Attribute access methods

Method	Description
`__getattr__(self, name)`	If this method is implemented, then it will be called with the attribute name when an attribute that *doesn't exist on the object* is requested. If fetching the attribute fails, it should raise an `AttributeError`.
`__setattr__(self, name, value)`	If this method is implemented, it will be called whenever an attribute is set on an instance. `__setattr__` can do "real" attribute setting by delegating to `object`: `object.__setattr__(self, name, value)`.
`__delattr__(self, name)`	If this method is implemented, it will be called whenever an attribute is deleted.
`__getattribute__(self, name)`	This method is always called for attribute access (new-style classes only). It is an important part of the descriptor protocol. It should return the attribute or raise an `AttributeError`. If both this method and `__getattr__` are explicitly defined, then `__getattr__` will be called only if `__getattribute__` calls it.

B.7 Numeric types

Numeric types are expected to implement a whole host of different methods in order to support the full range of numeric operations and conversions between the different built-in types. In addition, other types are free to implement individual methods to support a subset of the operations. For example, strings and lists both implement

the addition methods so that you can add strings and lists together. Strings implement the modulo operator for string interpolation.

B.7.1 Arithmetic operations

Table B.7 shows the methods that implement the binary arithmetic operations (+, -, *, /, //, true division, %, divmod(), pow(), **, <<, >>, &, ^, |). If one of those methods does not support the operation with the supplied arguments, it should return NotImplemented.

Table B.7 Binary arithmetic operations

Method	Description
`__add__(self, other)`	Called for addition operations (+).
`__sub__(self, other)`	Called for subtraction operations (-).
`__mul__(self, other)`	Called for multiplication (*).
`__floordiv__(self, other)`	Called for floor division (//).
`__div__(self, other)`	Called for division (/).
`__truediv__(self, other)`	Called for division when true division is on.
`__mod__(self, other)`	Called for the modulo operator (%).
`__divmod__(self, other)`	Called by the built-in divmod function. This method should be the equivalent to using `__floordiv__` and `__mod__`.
`__pow__(self, other[, modulo])`	Called for power operations (**). Can also be called by the built-in pow function. This method must accept an optional third argument if it is to support the three-argument form of the pow function.
`__lshift__(self, other)`	Called for left-shift operations (<<).
`__rshift__(self, other)`	Called for right-shift operations (>>).
`__and__(self, other)`	Called for binary and operations (&).
`__xor__(self, other)`	Called for binary exclusive or operations (^).
`__or__(self, other)`	Called for binary or operations (|).

These methods all have an additional two forms.

If they are on the right-hand side of a binary operation, then the right-hand version of the operator method will be called. This has the same name as the normal operator but with an r prepended. For example, the right-hand addition operator method is `__radd__`.

All of the operators (except for the divmod function) also have an in-place equivalent. The in-place operators are +=, -=, /=, //=, %=, **=, <<=, >>=, &=, ^=, and |=. The in-place operator methods have the same name as the standard operator methods

with an `i` prepended. For example, the in-place addition operator method is `__iadd__`. Because in-place operations rebind the name being operated on, mutable values that implement in-place operators should return `self` from the in-place operator methods (immutable values can return a new value as normal). If an in-place operator is used on an object that implements the standard operator method but *not* the in-place operator method, then the standard operator method will be used instead. Table B.8 shows the unary arithmetic operations.

Table B.8 Unary arithmetic operations

Method	Description
`__neg__(self)`	Called for the unary negation operator (`-`)
`__pos__(self)`	Called for the unary plus operatory (`+`)
`__abs__(self)`	Called by the built-in function `abs`; returns the absolute value
`__invert__(self)`	Called by the invert operator (`~`)

B.7.2 Conversion between numeric types

The methods in table B.9 are used to convert objects between numeric types, usually by built-in functions.

Table B.9 Type conversion

Method	Description
`__complex__(self)`	Called by the built-in function `complex`. Should return a complex number.
`__int__(self)`	Called by the built-in function `int`. Should return an `int`.
`__long__(self)`	Called by the built-in function `long`. Should return a `long`.
`__float__(self)`	Called by the built-in function `float`. Should return a `float`.
`__oct__(self)`	Called by the built-in function `oct`. Should return a `string` (the octal representation).
`__hex__(self)`	Called by the built-in function `hex`. Should return a `string` (the hex representation).
`__index__(self)`	Called whenever Python needs an integer object (such as in slicing). Must return an integer (`int` or `long`). New in version 2.5.

B.8 Context managers and the with statement

The Python `with` statement (new in Python 2.5)[6] supports the concept of a runtime context. This allows you to perform operations on an object, with a context manager

[6] Using `with` in Python 2.5 (IronPython 2) requires the future import: `from __future__ import with_statement`.

to handle closing or disposing of the object once the operations are complete. This is useful for committing database transactions (or rolling them back in case of an error), synchronizing threads, or working with files.

> ### with as RAII
> The Python `with` statement is analogous to the C# `using` statement and is a form of RAII (Resource Acquisition Is Initialization). `with` is an improvement on RAII in C++ because it can tell if an exit is exceptional or not.

The context management protocol involves implementing a pair of methods, `__enter__` and `__exit__`. Table B.10 describes these methods.

Table B.10 The context management protocol

Method	Description
`__enter__(self)`	Enter the runtime context of the context manager. The object(s) returned by this method are bound to the names in the `as` clause of the `with` statement (if any).
`__exit__(self, exc_type, exc_val, exc_tb)`	Exit the runtime context associated with the context manager. If the `with` statement exits normally, then `exc_type`, `exc_val`, and `exc_tb` will all be `None`. If an exception occurs, then they will have the same values as would be returned by `sys.exc_info()`. Context managers should not re-raise exceptions; instead they should return a Boolean indicating whether or not they have handled the exception (a return value of `False` will cause the exception to be re-raised after `__exit__` has completed).

Python defines these methods on files, so that you can use the following pattern:

```
with open(filename) as handle:
    data = handle.read()
    ...
```

When the `with` statement is executed,[7] `__enter__` is called and the file handle returned is bound to the name `handle`. When the code inside the `with` block is exited (or an exception is raised), `__exit__` is called, which closes the file.

The standard library module `contextlib`[8] is extremely useful for working with the context management protocol. `contextlib.closing` is a convenient decorator for creating context managers that automatically close a resource. When `closing` is insufficient, `contextlib.contextmanager` is the simplest way to create a context manager. When working with decorators in general, `functools`[9] is another extremely useful

[7] The `with` statement does not create a new scope.
[8] See http://docs.python.org/library/contextlib.html.
[9] See http://docs.python.org/library/functools.html.

module. For example, functools.wrap preserves the name (__name__) and docstrings of decorated functions.

B.9 The descriptor protocol

The descriptor protocol is an aspect of Python that is regarded as deep black magic, perhaps even more so than metaclasses. Like metaclasses, the rules are very simple once you understand them; it is just not very often that you need to explicitly implement them. The descriptor protocol is how properties, class methods, and static methods are implemented.

When an object is fetched from a class or instance dictionary, the __getattribute__ method checks for these methods and invokes them appropriately to fetch the object. This allows you to customize what happens when an object is accessed as a class or instance attribute. Table B.11 shows the descriptor protocol methods.

Table B.11 Descriptor protocol methods

Method	Description
`__get__(self, instance, owner)`	Called to get attributes from an instance or a class (`owner`). If fetched from a class, `instance` will be `None`.
`__set__(self, instance, value)`	Called to set the attribute on an instance of the `owner` class to a new value.
`__delete__(self, instance)`	Called to delete an attribute on an instance of the `owner` class.

Python has a built-in property descriptor that allows you to invoke code when an attribute is requested on an instance. It doesn't have an equivalent classproperty allowing you to create static properties as you can in C#. Using the descriptor protocol we can implement classproperty in a few lines of code:

```
class classproperty(object):
    def __init__(self, function):
        self.function = function

    def __get__(self, instance, owner):
        return self.function(owner)
```

classproperty can be used as a decorator. When it decorates a method (which should receive the class as its only argument), it replaces the method in the class with a descriptor. The original method is stored as a function object, the function attribute on the descriptor.

When the descriptor is fetched from the class, its __get__ method is invoked, and the stored function is called with the class passed in. Despite its mystery-shrouded reputation, the descriptor protocol can be used very simply to add new language features.

B.10 Magic attributes

As well as magic methods, many Python objects have magic attributes. These are attributes whose names start and end with double underscores and that have a special meaning to the Python interpreter. Table B.12 lists the most common of these attributes along with their description.

Table B.12 Python magic attributes

Name	Description
`__doc__`	Most Python objects have a `__doc__` attribute that holds the docstring for the object. If there is no docstring, this attribute will be `None`. Docstrings are displayed by the help function and used by automated documentation tools. Docstrings can be assigned directly or created with a string literal as the first entry (before any code) in a module, class, function, or method.
`__dict__`	`__dict__` is a read-only attribute that holds all members of an object. It forms the object namespace. It exists for all objects except for those that use `__slots__`.
`__slots__`	When `__slots__` is set as a class variable (a list of strings), it reserves space in the class for the named members. Instances of classes with `__slots__` will not have a `__dict__` dictionary, and attempting to set an attribute not listed in `__slots__` will raise an `AttributeError`. It is intended as a memory optimization, saving the space required for a dictionary per instance. In IronPython classes that define `__slots__` are heavily optimised; attribute lookup is approximately 4x faster. `__slots__` has several caveats about its use. See the Python documentation for more details.[a]
`__class__`	A reference to the class for all class instances (everything in Python is an instance of a class).
`__bases__`	A class attribute referencing a tuple of the base classes for the class.
`__name__`	An attribute (string) on modules, functions, and classes. It is assignable, allowing decorated functions to retain the same name as the function they wrap (very useful for tracebacks from decorated functions).
`__all__`	An optional attribute (list of strings) for modules. If available, this lists all the names that will be exported by the module when `from module import *` is executed. Defining `__all__` can be a useful way of defining the public API of a module.
`__file__`	A module attribute (string) containing the path the module was loaded from on the filesystem. This attribute does not exist for built-in modules.
`__module__`	A class, function, and method attribute (string) with the name of the module the object was defined in.
`__metaclass__`	If this is defined as a class attribute, then the callable assigned will be called to create the class instead of type. It can also be used as a module-level attribute, and all old-style classes (without an explicit alternative metaclass or inheriting from a class with a metaclass) will use the callable assigned as their metaclass. A quick way to convert all old-style classes in a module into new-style classes is to add `__metaclass__ = type` at the start of the module. (It doesn't affect new-style classes, as they inherit `type` as a metaclass from `object`.)
`__debug__`	A global variable (Boolean) indicating whether the interpreter is being run with optimizations on (`-O` or `-OO` command-line switches).

a. See http://docs.python.org/ref/slots.html.

B.11 Functions and modules

There are also a handful of modules and one built-in function whose names start and end with double underscores. Table B.13 lists the magic modules and functions.

Table B.13 Magic functions and modules

Name	Description
`__builtin__`	The Python module that contains all the built-in functions and exceptions. You can import this and patch it to affect the global scope of all modules. You can effectively create new built-ins, or point existing names to different objects, at runtime.
`__main__`	This module is the main script (as a module) that the interpreter is executing.
`__init__`	Python packages consist of directories containing an `__init__.py` file. As well as marking the directory as a package, this module acts as the top-level namespace for the package.
`__import__`	A built-in function that provides programmatic access to the import machinery. The function signature is `__import__(name[, globals[, locals[, fromlist[, level]]]])`. See the Python documentation on built-in functions[a] for more details.

a. See http://docs.python.org/lib/built-in-funcs.html.

appendix C: For more information

This appendix lists online sources of information about IronPython and dynamic languages on the .NET framework.

The most important website is the one for this book. The website contains news and the downloadable source code for the examples. In the unlikely event of any mistakes being found, it will also be a home for the errata.

- http://www.ironpythoninaction.com/

C.1 IronPython and Python language sites

- IronPython project page—http://www.codeplex.com/IronPython
- Python—http://www.python.org/
- Python documentation—http://docs.python.org/
- FePy, IronPython Community Edition—http://fepy.sourceforge.net/
- IronPython for ASP.NET—http://www.asp.net/DynamicLanguages/

C.2 Mailing lists and newsgroups

- IronPython mailing list—http://lists.ironpython.com/listinfo.cgi/users-ironpython.com
- Google Groups gateway—http://groups.google.com/group/ironpy
- Python newsgroup—comp.lang.python
- Google Groups gateway—http://groups.google.com/group/comp.lang.python/topics
- Python announce newsgroup—comp.lang.python.announce
- Google Groups gateway—http://groups.google.com/group/comp.lang.python.announce/topics

C.3 Python and IronPython code examples

- IronPython Cookbook—http://www.ironpython.info/
- ActiveState Python Cookbook—http://code.activestate.com/recipes/langs/python/
- Python Package Index—http://pypi.python.org/pypi

C.4 Learning Python

- Python tutorial—http://docs.python.org/tutorial/
- Dive Into Python—http://diveintopython.org/
- Michael Foord's IronPython articles—http://www.voidspace.org.uk/ironpython/
- Python quick reference—http://rgruet.free.fr/
- A Byte of Python—http://www.swaroopch.com/notes/Python

C.5 Blogs

- Michael's blog—http://www.voidspace.org.uk/blog
- IronPython URLs news and link aggregator—http://ironpython-urls.blogspot.com
- Planet Python—http://planet.python.org
- Planet IronPython—http://www.voidspace.org.uk/ironpython/planet/
- Jeff Hardy, developer of NWSGI—http://jdhardy.blogspot.com/

C.6 IronPython team

- Jim Hugunin, creator of IronPython—http://blogs.msdn.com/hugunin/
- Harry Pierson, IronPython PM—http://devhawk.net/
- Curt Hagenlocher, core developer—http://blogs.msdn.com/curth/
- Dino Viehland, core developer—http://blogs.msdn.com/dinoviehland/
- Jimmy Schementi, Silverlight & dynamic languages PM—http://blog.jimmy.schementi.com/
- Martin Maly, DLR core developer—http://blogs.msdn.com/mmaly/
- Dave Fugate, IronPython tester—http://knowbody.livejournal.com/
- Srivatsn Narayanan, IronPython tester—http://blogs.msdn.com/srivatsn/
- Oleg Tkachenko, IronPython and Visual Studio integration—http://www.tkachenko.com/blog/
- Seshadri Pillailokam Vijayaraghavan, DLR tester—http://blogs.msdn.com/seshadripv/default.aspx
- Shri Borde, IronPython/IronRuby dev lead—http://blogs.msdn.com/shrib/
- Bill Chiles, DLR PM —(Bill doesn't have a blog or website)

C.7 Silverlight

- Silverlight homepage—http://silverlight.net/
- Moonlight—http://www.mono-project.com/Moonlight
- Silverlight Dynamic Languages SDK—http://www.codeplex.com/sdlsdk
- Michael Foord's Silverlight articles—http://www.voidspace.org.uk/ironpython/silverlight/

C.8 .NET and Mono

- .NET framework—http://msdn.microsoft.com/en-us/netframework/default. aspx
- Mono homepage—http://www.mono-project.com/Main_Page
- C# 4.0 and Visual Studio 10 CTP—http://go.microsoft.com/fwlink/?LinkId=129231

C.9 Dynamic languages on .NET

- Dynamic Language Runtime project page—http://www.codeplex.com/dlr
- Python.NET, CPython .NET integration—http://pythonnet.sourceforge.net/
- IronRuby—http://ironruby.net/
- IronScheme—http://www.codeplex.com/IronScheme
- Boo, a Python-inspired statically typed .NET language with duck typing—http://boo.codehaus.org/
- Cobra, another Python-inspired statically typed .NET language—http://cobra-language.com/

C.10 IDEs and tools

- SharpDevelop—http://www.icsharpcode.net/OpenSource/SD/Default.aspx
- MonoDevelop—http://monodevelop.com/Main_Page
- Wing IDE—http://www.wingware.com/
- Visual Studio—http://www.microsoft.com/VisualStudio/default.mspx
- Visual Studio Express—http://www.microsoft.com/Express/
- IronPython Studio—http://www.codeplex.com/IronPythonStudio
- Resolver One, IronPython spreadsheet—http://www.resolversystems.com/
- Crack.NET, debugger with IronPython scripting—http://www.codeplex.com/cracknetproject

index

Symbols

A

D

O

P

Q

R

S

Microsoft .NET

IronPython IN ACTION

Michael J. Foord • Christian Muirhead • Foreword by Jim Hugunin

IronPython, Microsoft's implementation of the Python programming language for .NET and Mono, combines Python's flexibility with easy access to the .NET libraries. IronPython enables diverse programming techniques such as functional and metaprogramming, and the Dynamic Language Runtime ensures compatibility with new technologies like Silverlight. The newly released IronPython version 2.0 is ready for prime time.

IronPython in Action introduces IronPython as a first class .NET language. You'll start by using the interactive console to explore the .NET framework with live objects. Following numerous small examples, you'll dive into the world of dynamic programming, including live introspection, dynamic and "duck" typing, metaprogramming, and more. Then you'll learn to access WPF, Silverlight, and other core .NET technologies. You'll even tackle advanced topics like embedding IronPython as a ready-made scripting language into C# and VB.NET programs.

This book assumes some programming experience but requires no previous knowledge of Python or .NET.

What's Inside

- An introduction and language tutorial
- Overview of .NET for Python programmers
- Embedding IronPython in C# applications
- Web programming with ASP.NET, Silverlight, and the DLR

About the Authors

A prolific speaker and writer, **Michael Foord** is a Python and .NET developer with Resolver Systems. Coauthor **Christian Muirhead**, also at Resolver Systems, is actively developing a next-generation data modeling tool in IronPython.

For online access to the authors, code samples, and a free ebook for owners of this book, go to www.manning.com/IronPythoninAction.com

Free ebook
SEE INSERT

"... everything you need to get started with IronPython."
—From the Foreword by Jim Hugunin, Creator of *IronPython*

"The best book on one of the best implementations of Python."
—Alex Martelli, Author of *Python in a Nutshell*

"A long-awaited guide"
—Keith J. Farmer Development Lead, Idea Entity

"... excellent explanations of unique and valuable features."
—Noah Gift, Author of *Python for Unix* and *Linux System Administration*

"If you are a .NET programmer this book is for you... and if you are new to programming, it's for you too."
—Craig Murphy, Microsoft MVP craigmurphy.com

ISBN-13: 978-1-933988-33-7
ISBN-10: 1-933988-33-9
54499
9 781933 988337

MANNING $44.99 / Can $44.99 [INCLUDING eBOOK]